Chapter 5

Stochastic Process Algebras

5.1 Introduction

In recent decades parallel and distributed systems have been extensively used in many application areas, and there has been significant effort in developing formal methodologies for the specification and analysis of such systems; unfortunately not always completely satisfactorily. The presence of concurrency, communication, synchronisation and nondeterminism makes the study of the correctness of concurrent systems particularly difficult, especially when the process interaction structure is not regular.

We have already presented in the previous chapters two approaches to modelling complex systems. Here we introduce a new one, known as *stochastic process algebras*, which provides a formal apparatus to reason about the qualitative and quantitative behaviour of these systems.

Following the style of previous chapters the presentation will be rather informal: we first start from the basics of the formalism and then describe some of the extensions which have been recently proposed in the literature to make it suitable for performance modelling. Stochastic process algebras, as was the case for the previously presented formalisms, are suitable for both the modelling and the analysis of discrete event dynamic systems, but here we are mainly concerned with the modelling aspects, referring to chapters which will appear later in this book for discussion of the analysis techniques. In the second part of the chapter we briefly discuss the relationship between stochastic process algebras and stochastic Petri nets.

5.2 Process Algebras

Process algebras (PA) are abstract languages which have been introduced for the specification and understanding of complex systems characterised by *communication* and *concurrency* [35]. These systems are seen as collections of entities that are acting independently most of the time, and which sometimes have to interact through communication in order to achieve a common goal. Process algebras, as we will discuss, provide a formal framework in which such systems can be defined, interpreted, and analysed.

5.2.1 Classical process algebras

Throughout this chapter we will refer to process algebras which have not been extended with notions of probability or time, as *classical process algebras*. Here we briefly overview the fundamentals of process algebra in terms of two classical process algebras which have been strongly influential in the field: CCS and CSP.

In a process algebra concurrent systems are described as collections of entities, called *agents*, that execute atomic *actions* which constitute the building blocks of the language. Complex systems are built starting from these building blocks and by applying the constructors of the algebra. In addition to the compositional operators, mechanisms for abstraction are provided which allow internal details to be disregarded. Legal terms in the algebra are defined by a grammar. The actions can be visible or not. We assume there is a set of visible actions, denoted Act. By convention, actions are represented by lower case letters.

The Calculus of Communicating Systems (CCS) [35] is considered the starting point in the field of PA. In CCS, Act consists of two sets, $\Lambda = \{a, b, c, \ldots\}$, the set of names, and $\overline{\Lambda} = \{\overline{a}, \overline{b}, \overline{c}, \ldots\}$, the set of co-names. An additional action, called the *silent* or *perfect* action τ, is introduced to represent internal (not visible) activity. The syntax of CCS is defined by the following grammar:

$$P ::= Nil \mid a.P \mid P + P \mid P|P \mid P \setminus L \mid P[f] \mid rec(X = P)$$

where $a \in \mathcal{A}ct \cup \tau$, $L \subseteq \mathcal{A}ct$, and X is a process variable.

The names and the intuitive meanings of the operators are the following:

- **Inactivity** Nil represents the agent that cannot perform any action.
- **Prefix** This operator constitutes the basic mechanism by which the behaviours of components are constructed: $a.P$ is the agent that can perform an action a and subsequently behaves like P.
- **Choice** $P + Q$ is a composite process that can behave either like P or like Q. When one component is selected the other is discarded, the choice being performed nondeterministically.
- **Parallel Composition** $P|Q$ is a process whose components P and Q might proceed concurrently and independently or they might synchronise. Two parallel agents can engage in a communication when one agent performs an action, say a, and the other agent performs complementary action $\overline{a}$. The result of the communication between the two partners is a τ action which is no longer visible in the environment.
- **Restriction** $P \setminus L$ represents the process that cannot perform any actions $\{a, \overline{a}\} \in L$.

- **Relabelling** $P[f]$ is an agent that behaves like P, but with the actions relabelled by the function f.

- **Recursion** The recursive term $rec(X = P)$ may describe an infinite behaviour: for example, the expression $rec(X = a.X)$ may be read as *"the agent X such that $X = a.X$"* and represents an agent that can perform an infinite number of actions a. In this so-called *rec* expression, X is a bound variable and can be renamed without any semantic effect.

An alternative way for describing infinite behaviours makes use of *constants*, i.e. names that can be assigned to patterns of behaviour associated with components. It is possible to specify equations like $A \overset{def}{=} P$, which gives the constant A the behaviour of the component P; if the definition of P contains A then an infinite behaviour is obtained. In the following we will use constant names instead of the *rec* operator.

Another well known process algebra is the Communicating Sequential Processes (CSP) originally introduced in [28] and in a more abstract version in [40]. In contrast to CCS, this language does not include the notion of complementary actions and a new form of communication between components is proposed. Together these allow the specification of multiway synchronisations rather than the strict pairings which are allowed in CCS.

The expression $P\|_S Q$ represents the parallel composition of P and Q with respect to the set S of joint actions (τ cannot belong to S). $P\|_S Q$ behaves like P or Q running independently of each other except for all actions contained in the set S on which they *must* synchronise and communicate. By varying the synchronisation set S, parallel composition $\|_S$ ranges from arbitrary interleaving, when S is the empty set (in this case the concise notation $P\|Q$ is usually used), to full synchrony, when S comprises all the possible actions. During the communication the joint action remains visible to the environment and it can be reused by other concurrent processes. This rule leads to a new kind of communication (multiway synchronisation) in which more than two processes can be involved.

In CSP it is also possible to abstract from internal details using a new operator called *hiding*. The process P/b represents a process that behaves like P except for the action b that is hidden from the external environment. The hiding operator differs from the CCS restriction operator. Restriction actually stops the actions with a given label from occurring, whereas hiding allows the actions to proceed invisibly.

CSP provides several other operators which will not be explained in this introductory description. The operators which have been presented here form the core set of operators found in most process algebras. In the following section we will introduce the process algebra which we will use for the remainder of this chapter, which contains elements from both CCS and CSP.

5.2.2 A process algebra: syntax and formal semantics

In this subsection we define the process algebra on which we will base the remainder of our presentation. As well as the syntax and semantics of this language we discuss some important features of process algebra, namely compositionality and abstraction mechanisms, as exemplified in our language.

Syntax As already explained the language we will use is an amalgamation of features from CCS and CSP. In particular we use some CCS operators plus the CSP synchronisation and hiding mechanisms. The motivation for choosing such a language is mainly that some PA extensions that we will discuss later in the chapter use this set of operators. Additionally, we feel that this language contains a sufficient set of operators to be expressive whilst not overawing the reader with details.

The grammar of our PA language is:

$$P ::= Nil \mid a.P \mid P+P \mid P\|_S P \mid P/L \mid A$$

where $a \in \mathcal{A}ct \cup \tau, S \subseteq \mathcal{A}ct, L \subseteq \mathcal{A}ct$. The intuitive meanings of the operators coincide with those given in the previous section (Section 5.2.1).

Once we have defined our set of language operators we need to associate a precise meaning with each of them, i.e. we need to define their semantics.

Interleaving semantics The formal semantic rules for the operators, which are usually presented in the Structural Operational Semantics (SOS) style of Plotkin [44], allow one to associate with each agent a Labelled Transition System (LTS). In general a LTS = $(S, T, \rightarrow)$ is defined by a set of states S, a set of transition labels T and a transition relation $\rightarrow \subseteq S \times T \times S$. In the case of PA the set of states is given by the set of language terms, the set of transition labels is given by the set of (atomic) actions, and the transition relation is given by the operational rules.

An example of SOS may be found in Figure 5.1 where the rules for the prefix and parallel composition operators are shown. These rules are read as follows: if the transition above the line is possible, then we can infer the transition below the line.

No precondition is necessary for prefix and the term $a.P$ can evolve into P by executing action a. Three different rules are needed to specify precisely the meaning of parallel composition. The first two rules define the behaviour of P and Q when executing independent actions. If P, by executing a, evolves into P' and $a \notin S$, then the composite process $P \|_S Q$, by executing a, can evolve into $P' \|_S Q$ (similarly for Q). The third rule defines the communication between the two components when executing an action in the synchronisation set S. If P executes a and Q executes a, an interaction can take place. The resulting activity is a joint action a and the whole process evolves into $P' \|_S Q'$.

By applying the semantic rules it is possible to associate with each language expression a *transition tree* that can be viewed as a way of describing the be-

Prefix

$$\frac{}{a.P \xrightarrow{a} P}$$

Parallel Composition

$$\frac{P \xrightarrow{a} P'}{P \parallel_S Q \xrightarrow{a} P' \parallel_S Q}\ a \notin S \qquad \frac{Q \xrightarrow{a} Q'}{P \parallel_S Q \xrightarrow{a} P \parallel_S Q'}\ a \notin S$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel_S Q \xrightarrow{a} P' \parallel_S Q'}\ a \in S$$

Figure 5.1: Prefix and parallel composition semantic rules.

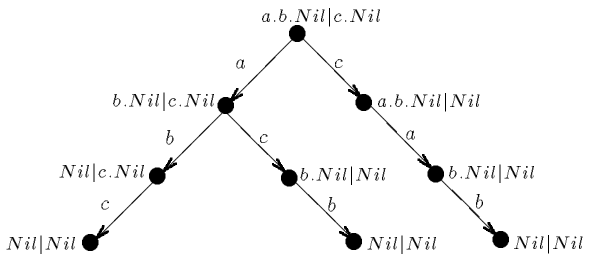


Figure 5.2: Transition tree of P.

haviour of the modelled system. The nodes of the tree are labelled with language expressions and the arcs with the actions that cause their evolution.

An example of a transition tree for the term $P = a.b.Nil \parallel c.Nil$ is shown in Figure 5.2. This tree corresponds to the interleaving semantics of P and gives all the possible sequences of actions that could be observed during the evolution of the model. Notice that in the interleaving semantics we abstract from the fact that the system is composed of separate components (only two in this simple example, $a.b.Nil$ and $c.Nil$) because we consider a *global* state without considering its distributed nature. Actions of independent components are merged with actions of the others in such a way that all possible interleavings are represented.

Some nodes in the tree of Figure 5.2 are labelled with the same syntactic expression. When we consider an equivalence notion between nodes for which the nodes characterised by the same label are considered to be equivalent, we can collapse them into a single one and obtain the *transition diagram* (also called *derivation graph*) of Figure 5.3. This notion of syntactic equivalence is

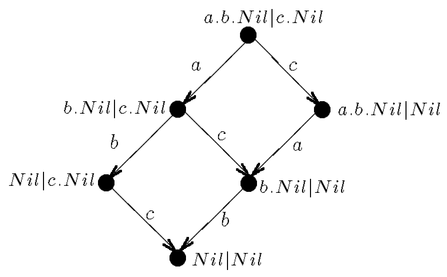


Figure 5.3: Transition diagram of P.

quite simple, probably the simplest notion of equivalence that could be defined. In the theory of PA significant effort has been given to the study of equivalence relations between states and between processes, and we will present some of them later in this chapter. We end this discussion by inviting the reader to observe the close relationship between the reachability graph underlying a net model and the derivation graph underlying a process algebra model.

Concurrent semantics A major drawback of interleaving models is that they abstract from the independence of the actions executed in independent subsystems. Noninterleaving models capture the fact that a system consists of a set of (partially) independent components and do not refer to the notion of a global state. The system's behaviour is modelled in terms of sequences of actions which are not required to be totally ordered, but which are only partially ordered. This partial order reflects the causal dependences between actions.

There is a strong debate around the choice of interleaving versus noninterleaving models for the definition of the semantics of PA and this chapter is not the right place to discuss this problem. We simply recall that some examples of noninterleaving models are Petri nets, event structures [52], and partial order traces [33] and that there is a branch of the research in this field related to the definition of a concurrent semantics for PA by means of noninterleaving models [9, 18, 41, 48, 31, 43].

Compositionality Process algebra models have been used extensively to establish the correct behaviour of complex systems [10, 6, 1]. These models are built following a *compositional* approach which constitutes a central feature of model construction.

Each subsystem is modelled in isolation and then the submodels are composed using the operators provided by the calculus in order to obtain the model of the whole system. Model components can be developed by different modellers and libraries of re-usable components may be established.

This leads to a hierarchical approach to model construction: the resulting model has a structure which reflects the structure of the system itself, is easy to understand, and readily modifiable. Moreover, this structure may be exploited during analysis.

The profound benefits of compositionality become apparent when we consider equivalence relations over models. If the relation can be shown to be a *congruence*—meaning that it respects the operators of the language in such a way that equivalence of components can be considered in isolation—analysis of the model via the relation can be carried out component by component. In general, this greatly reduces the complexity of the models which need to be tackled, during model verification for example.

Abstraction mechanism The abstraction mechanism allows some behaviour of the system to be abstracted away, for example because it is more detailed than necessary for the current model. In our PA this is provided by the hiding operator which allows us to abstract some aspect of the behaviour of a component so that it is not visible to an external observer, or to other components. The component P/L in fact behaves as P, except that the activities of types within the set L are *hidden*. They appear as the silent type τ and they are considered to be internal to the component in which they occur. Thus hiding allows an interface to a model or component to be defined.

This is particularly powerful when used in conjunction with the parallel composition combinator as it may restrict the interactions of a component. Components of a system may be modelled individually in detail, but subsequently in a more abstract form as the interactions between them are developed. For example, let us consider the term $P = a.b.Nil$ which offers the two actions a and b to the environment. By hiding one of the two actions, say a, we have $P' = P/a$ and we change the system interface since now P' offers only b. This means that if we want to compose P' with other components, or with new replicas of P, we can only require synchronisation on b, the action a now being internal.

5.2.3 The PLC multicomputer example

To better understand how PA may be used to describe system behaviours we now consider the simple multicomputer PLC example discussed earlier in the book (see Example 2.1 in Chapter 2). We recall that the example consists of a multicomputer architecture responsible for the control of a distributed plant. Each computer has a double access memory which is connected to a common bus. The behaviour of the PLC is cyclic: first the computers synchronise to start a control cycle, then they perform their calculation independently as long as they need to read external data.

The actions executed by the PLC components form the set:

$$\begin{aligned} Act_{PLC} = \; & \{start\_cycle, local\_comp, end\_cycle, acq\_bus, read\_ext\_data, \\ & \qquad req\_ext\_data\} \end{aligned}$$

The entities involved in the system are first modelled in isolation. Each computer is a *sequential* component whose behaviour may be expressed by means

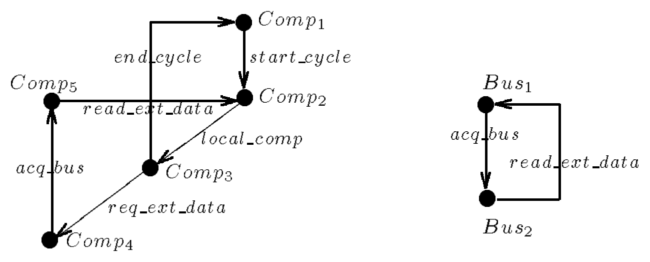


Figure 5.4: Transition diagrams of the computer (left) and the bus (right) components.

of the prefix and choice operators as follows:

$$\begin{aligned}
Comp_1 &= start\_cycle.Comp_2 \\
Comp_2 &= local\_comp.Comp_3 \\
Comp_3 &= end\_cycle.Comp_1 + req\_ext\_data.Comp_4 \\
Comp_4 &= acq\_bus.Comp_5 \\
Comp_5 &= read\_ext\_data.Comp_2
\end{aligned}$$

The bus is acquired and used for reading external data; a possible specification could be the following:

$$\begin{aligned}
Bus_1 &= acq\_bus.Bus_2 \\
Bus_2 &= read\_ext\_data.Bus_1
\end{aligned}$$

By applying the semantic rules we can derive the transition diagrams of $Comp_1$ and Bus_1 which are shown in the left and right parts of Figure 5.4 respectively.

After having specified the behaviour of the single entities, we need to express the behaviour of the complete system. This is done by composing the entities and by establishing their interactions.

As in the Example 2.1 discussed in Chapter 2, let us suppose the system is formed by two computers which synchronise to start a control cycle. This system is modelled by two instances of the entity $Comp_1$ which synchronise on the action *start_cycle*, representing the beginning of the control cycle. Moreover, both computers need the bus in order to read external data. This behaviour is achieved by adding one instance of the Bus_1 component with an appropriate synchronisation set. The specification of the PLC example is thus the following:

$$PLC = (Comp_1 \|_{\{start\_cycle\}} Comp_1) \|_{\{acq\_bus, read\_ext\_data\}} Bus_1$$

Once we have specified the complete model, we can derive the underlying transition diagram. Investigation of this diagram may prove several qualitative properties of the system as we will see in Chapter 6 when discussing the analysis methods based on the construction of the reachability (derivative) graph. Figure 5.5 shows a portion of the transition diagram underlying the PLC specification where, for simplicity, we have used short labels for the states (the name

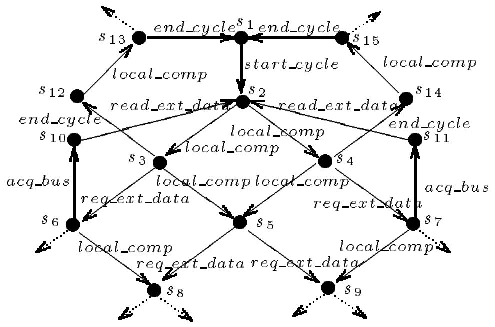


Figure 5.5: Portion of the transition diagram underlying the PLC example.

s_1	$(Comp_1 \|_S Comp_1) \|_L Bus_1$
s_2	$(Comp_2 \|_S Comp_2) \|_L Bus_1$
s_3	$(Comp_3 \|_S Comp_2) \|_L Bus_1$
s_4	$(Comp_2 \|_S Comp_3) \|_L Bus_1$
s_5	$(Comp_3 \|_S Comp_3) \|_L Bus_1$
s_6	$(Comp_4 \|_S Comp_2) \|_L Bus_1$
s_7	$(Comp_2 \|_S Comp_4) \|_L Bus_1$
s_8	$(Comp_4 \|_S Comp_3) \|_L Bus_1$
s_9	$(Comp_3 \|_S Comp_4) \|_L Bus_1$
s_{10}	$(Comp_2 \|_S Comp_5) \|_L Bus_2$
s_{11}	$(Comp_5 \|_S Comp_2) \|_L Bus_2$
s_{12}	$(Comp_1 \|_S Comp_2) \|_L Bus_1$
s_{13}	$(Comp_1 \|_S Comp_3) \|_L Bus_1$
s_{14}	$(Comp_2 \|_S Comp_1) \|_L Bus_1$
s_{15}	$(Comp_3 \|_S Comp_1) \|_L Bus_1$

Table 5.1: Labels of the states in the transition diagram of Figure 1.4 (where $S = \{start\_cycle\}$, $L = \{acq\_bus, read\_ext\_data\}$).

of each state is listed in Table 5.1). The complete transition diagram has 24 states and 47 transitions.

Let us now suppose that we want to add new computers to our system. This extension may be achieved compositionally by combining more instances of the $Comp_1$ component. For example, in the case of four computers and one common bus we have:

$$\begin{aligned} PLC' = \ & (Comp_1 \|_{\{start\_cycle\}} Comp_1 \|_{\{start\_cycle\}} Comp_1 \|_{\{start\_cycle\}} Comp_1) \\ & \|_{\{acq\_bus, read\_ext\_data\}} Bus_1 \end{aligned}$$

Finally, let us suppose that we do not want to observe the acquisition of the global bus, but only the data exchange. We can model this situation by taking advantage of the abstraction mechanism and by hiding the action acq_bus:

$$\begin{aligned} PLC'' = \ & ((Comp_1 \|_{\{start\_cycle\}} Comp_1 \|_{\{start\_cycle\}} Comp_1 \|_{\{start\_cycle\}} Comp_1) \\ & \|_{\{acq\_bus, read\_ext\_data\}} Bus_1)/\{acq\_bus\} \end{aligned}$$

5.2.4 Modelling features

Causal dependences, Concurrency, and Conflicts Bearing in mind the informal meanings of the algebraic operators, let us briefly discuss now the adequacy of the formalism for modelling causal dependences, concurrency, and conflicts between actions, modelling features that we have already discussed in Chapter 2 in the context of PN models.

Causal dependences appear in the form of sequences of actions which can be easily modelled by means of the prefix operator. In the PN context we said that a transition t_j follows a transition t_i when they are connected through a place. In the PA context we can say that an action b follows an action a when they are in the form $a.b$. Paraphrasing the net terminology, we could say that *"a and b are connected through the prefix operator."*

Concurrency between actions is syntactically represented by means of the parallel composition operator. For instance, $a.Nil \,||\, b.Nil$ denotes an expression in which the two actions a and b are simultaneously enabled and independent from each other. However, as we have already discussed, when we consider the interleaving semantics of this term, we have that a and b can occur in any order, first a and then b, or vice versa, but never simultaneously.

Moreover, in the interleaving approach causality information may be lost. Let us consider, for example, the two agents $P = a.Nil \,||\, b.Nil$ and $Q = a.b.Nil + b.a.Nil$. Under the interleaving assumption the external behaviour of P and Q is the same: an external observer will either see first an a and then a b, or first a b and then an a, and therefore will not be able to distinguish between them. The observer is not able to detect that in $a.Nil \,||\, b.Nil$ the actions a and b occur independently, while in $a.b.Nil + b.a.Nil$ there is a causal dependence between them: a occurs before b or b occurs before a. In contrast, in the noninterleaving approach to semantics, these two agents are no longer indistinguishable because a distinction occurs between P, in which a and b can occur simultaneously, and Q in which this option is not present.

Conflicts in sequential systems represent situations in which two or more actions are enabled but only one can occur. The basic mechanism to model conflicts in PA makes use of the alternative or choice operator: the expression $P = a.Nil + b.Nil$ models a situation in which both actions a and b are enabled but only one of them can occur, the choice being performed nondeterministically. This simple form of conflict can be assimilated to the free-choice conflict already discussed. When we move to concurrent systems things become more complicated. Consider again the term P and suppose it is composed in parallel with another term Q, requiring that they have to synchronise on one of the two actions, for example a (i.e. $P\|_{\{a\}}Q$). Depending on the specification of Q we can have a conflict or not: indeed, if Q does not offer any action a, the choice will be always resolved in favour of b. The reader is invited to observe the close relation between this situation and the non free-choice conflicts of PN.

Structure and state As already discussed in Chapter 2, a PN model of a dynamic system consists of a net structure and a marking and it is possible to reason at two different levels, structural and behavioural.

There is no syntactic distinction between structure and state in PA. However, at the semantic level, in the LTS, a state is associated with each syntactic term. Thus for a PA model each derivative of the initial expression representing the model is considered to be a state. Observe that, since the model and each derivative are specified in the algebraic language, it is impossible to distinguish syntactically whether a term is a derivative of a model or a model itself.

Model analysis Due to the lack of an explicit structure it is not generally possible to prove properties which hold for any initial state, as in the case of PN models. Qualitative properties, like the absence of *deadlock* or the *reachability of a given state*, are then investigated by inspecting the transition diagram associated with the model as will be discussed in detail in Chapter 6. Recent work by Gilmore *et al.* [16] has established a structural theory for PA and this is currently under development.

5.2.5 Equivalence notions and equational laws

One of the most attractive features of PA is their compositional nature, but it is not the only one. Another important aspect of the formalism is the definition of equivalence relations [35], which can be used to compare agents (*model verification*) and to replace one agent by another which exhibits an equivalent behaviour, but has a simpler representation (*model simplification*). Such notions of equivalence are considered part of the semantics of the language, and therefore their definition is an integral part of its development. Another use of equivalence relations is over the states within a model. When a set of states are found to have equivalent behaviour we can simplify analysis by using the relation to partition the state space and considering only one representative of each partition (*model aggregation*). This is an important means of state space reduction.

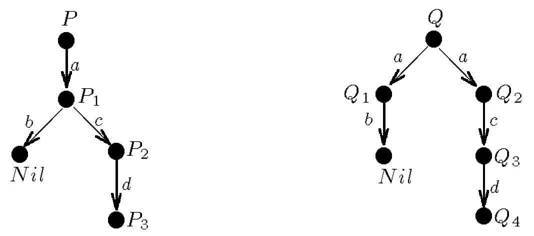


Figure 5.6: Transition diagrams of the two agents P (left) and Q (right).

We have already briefly discussed the notion of syntactic equivalence between the states forming the transition tree of a model. Unfortunately this relation has no benefits for model aggregation since the number of states remains unchanged. However, other equivalence notions between states have been proved to be very powerful for model aggregation and will be discussed in Chapter 16 while here we limit the discussion to the equivalence notions between agents.

The semantic theory of PA is an observational theory: the behaviour of a system corresponds to what is observable, and to observe a system corresponds to communicating with it. This approach is based on a single observer (interleaving) model in which the occurrence of events is serialised even for independent processes. Let us consider the agents:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
P & = & a.P_1 & Q & = & a.Q_1 + a.Q_2 \\
P_1 & = & b.Nil + c.P_2 & Q_1 & = & b.Nil \\
P_2 & = & d.P_3 & Q_2 & = & c.Q_3 \\
 & & & Q_3 & = & d.Q_4
\end{array}$$

and imagine an observer R trying to interact with them, i.e. $R\|_S P$ and $R\|_S Q$, with $S = \{a, b, c, d\}$. At the beginning both agents P and Q offer a to R (see their transition diagrams in Figure 5.6). But after the action a is performed, a difference emerges between them. P is deterministic: after the execution of action a, it always evolves into P_1 and it subsequently offers to the observer both actions b and c. The agent Q is instead nondeterministic due to the presence of the choice operator in its description. After the execution of the action a, Q can evolve into Q_1 *or* into Q_2 and it offers to the observer sometimes an action b and sometimes an action c. Thus, even though P and Q offer the same set of actions to R, they do not exhibit an equivalent behaviour.

We need to define some criteria to decide when two agents can be considered equivalent. As the meaning of agents is expressed by their corresponding transition diagrams we could think of two agents being equivalent when these graphs are isomorphic. However, isomorphism has been proved to be too strong a requirement and new notions of equivalence have been proposed [35]. In CCS for example, two agents are considered to be equivalent when their externally observed behaviours appear to be the same or, in other words, when any action by one can be matched by an action of the other and afterwards they remain equivalent. This notion of equivalence is known as *strong bisimulation*; it was first introduced in [42] and formally defined for CCS in [35] and it is fundamental in the PA theory.

Definition 5.1 *A binary relation $\mathcal{R}$ over agents is a* **strong bisimulation** *if* $(P, Q) \in \mathcal{R}$ *implies, for all* $a \in \mathcal{A}ct \cup \tau$:

1. *Whenever* $P \xrightarrow{a} P'$, *then for some* Q', $Q \xrightarrow{a} Q'$ *and* $(P', Q') \in \mathcal{R}$;
2. *Whenever* $Q \xrightarrow{a} Q'$, *then for some* P', $P \xrightarrow{a} P'$ *and* $(P', Q') \in \mathcal{R}$.

Any relation which satisfies Definition 5.1 is a strong bisimulation. The notion of *strong equivalence* ($\sim$), also called *strong bisimilarity*, is then introduced as the largest strong bisimulation.

To establish a bisimilarity between two agents P and Q it is necessary to set up a correspondence between the states in the respective transition diagrams. We may build such a correspondence starting with the pair (P, Q) and seeing which pairs must be in correspondence given that (P, Q) is such a pair. The procedure is then repeated with each of the new pairs so introduced until no more new pairs exist. In the end, if all the states in the transition diagrams are contained in some pair, it can be concluded that the two agents are strongly bisimilar.

The agents P and Q described earlier (see Figure 5.6) do not satisfy Definition 5.1. When we start from the pair of initial states (P, Q), both processes can execute action a evolving into P_1 and Q_1 or Q_2. The new pairs (P_1, Q_1) and (P_1, Q_2) seem to be the candidates for setting up the correspondence. However, P_1 and Q_1 (resp. P_1 and Q_2) are not equivalent because they do not offer the same actions. The computation stops and, since not all the states belong to some pair, we can conclude that the two agents are not strongly bisimilar.

As already mentioned, another fundamental notion in the PA theory is the notion of congruence. An equivalence relation is a congruence when it is preserved by all combinators of the language. Strong equivalence is a congruence and therefore allows the full advantages of compositionality. In fact, whenever two agents P and Q are strongly equivalent, they can be interchanged in any complex system S, with confidence that its behaviour remains the same. This means that model verification and model simplification can be carried out on the components within a model, rather than across the whole model at once.

Strong bisimilarity is rather restrictive because every action an agent may perform must be matched by an action of the same type in the equivalent agent — even τ actions. However, we would like to consider equivalent terms like $a.Nil$ and $a.\tau.Nil$ since their observable behaviours are the same: they both execute an action a and then terminate. The latter term in fact has an internal state change that we wish to consider invisible. For this purpose, a weaker notion of bisimulation, called *weak bisimulation* ($\approx$), has been defined [35]. In this case it is required that each τ action in one term must be matched by zero or more τ actions in the other term. Under weak bisimulation it can be proved that $a.Nil \approx a.\tau.Nil$ (while $a.Nil \not\sim a.\tau.Nil$). On the other hand, $\approx$ is not preserved by the choice operator: when we compose an agent in choice with two terms which are equivalent under weak bisimulation, we do not necessarily obtain two expressions which are still equivalent. For instance we have:

$$b.Nil \approx \tau.b.Nil \quad \text{but} \quad a.Nil + b.Nil \not\approx a.Nil + \tau.b.Nil$$

because the second term can autonomously (by executing τ) reach a state in which only b is possible while the first agent must always choose between a and b. This fact implies that $\approx$ is not a congruence. However, a refinement of $\approx$, *observational congruence*, is known to have this desirable property.

A lot of work has been done in the definition of different equivalence notions and we refer the reader to the literature (see for example [49, 50]) for an extensive discussion. Chapter 16 also provides more insight into this topic.

Equational Laws The equivalence notions allow the proof of a set of equational laws that can be used to manipulate the language expressions. If an algebraic characterisation of the equivalence (axiomatisation) is found, the axioms may be used to apply it at the level of the syntax rather than at the level of the underlying transition system. We do not intend to give a complete list of all the equational laws here, but we show some of them below to give an idea of the algebraic axiomatisation of the strong bisimulation relation. More details can be found in the literature; for example, the complete list of the CCS laws can be found in [35].

$$\begin{array}{ll} 1)\ P+Q \sim Q+P & 2)\ P \parallel_S Q \sim Q \parallel_S P \\ 3)\ P+Nil \sim P & 4)\ P \parallel_S Nil \sim P \\ 5)\ P+P \sim P & \end{array}$$

The equations 1) and 2) show the commutativity of the " $+$ " and " $\parallel_S$ " operators. Equations 3) and 4) are related to the Nil operator: no external observer can detect if an agent is composed with the Nil agent using " $+$ " or " $\parallel_S$ ". The last law, $P + P \sim P$, states that no external observer can detect if the agent he is observing is composed with one or more copies of itself in a nondeterministic choice: the two agents $P + P$ and P cannot be distinguished in terms of the actions they offer to the environment.

The most important law is called the *expansion principle* or *expansion law* and it expresses the behaviour of a complex system by means of the behaviours of the composite subsystems. Generally, due to the interleaving semantics, the parallel composition of a finite number of agents can be transformed into an equivalent specification in which parallel composition is replaced by the choice and the prefix operators. For example the term $P = a.b.Nil \parallel c.Nil$ can be rewritten as $P = a.b.c.Nil + a.c.b.Nil + c.a.b.Nil$.

In the case of a complex system $P = P_1 \parallel P_2 \parallel \ldots \parallel P_n$ in which each component P_i can proceed independently the overall behaviour of P can be seen as follows:

$$P_1 \parallel P_2 \parallel \cdots P_n \ \sim\ \sum\{a.(P_1 \parallel \cdots \parallel P_i' \parallel \cdots \parallel P_n) : 1 \leq i \leq n, P_i \xrightarrow{a} P_i'\}$$

In its most general form the expansion law is more complicated and we refer the reader to the literature for more details.

Notice that the expansion law makes interleaving between actions explicit since all possible sequences of actions are represented.

When an equivalence relation has been identified which is useful for model simplification, if an axiomatisation exists it opens the possibility of the simplification procedure being automated via a rewriting system, see for example [29].

5.2.6 The PLC multicomputer example revisited

Let us consider again the PLC model presented in Section 5.2.3. Using axiom 2) from the previous section we can see that the derivatives:

$$(Comp_2 \|_{\{start\_cycle\}} Comp_4) \|_{\{acq\_bus, read\_ext\_data\}} Bus_1$$
$$(Comp_4 \|_{\{start\_cycle\}} Comp_2) \|_{\{acq\_bus, read\_ext\_data\}} Bus_1$$

are strongly bisimilar, because

$$(Comp_2 \|_{\{start\_cycle\}} Comp_4) \sim (Comp_4 \|_{\{start\_cycle\}} Comp_2)$$

and $\sim$ is a congruence. Intuitively we can see that this makes sense, from the point of view of an external observer, if one computer is engaged in local computation ($Comp_2$) and the other is trying to acquire the bus ($Comp_4$) it does not matter which computer is in which state because the observable behaviour of the system will be the same in either case.

5.3 Probabilistic Process Algebras

Various extensions of PA have been proposed in the literature. We briefly present here *probabilistic process algebras* [30, 32, 45], a class of languages in which nondeterministic choice has been replaced by probabilistic choice to capture uncertainty about the behaviour of the modelled system. A new operator, let us use the notation $\oplus$, has been introduced for specifying the probabilistic choice between two or more alternatives. For example, the expression $P = a \oplus_p b$ represents a process that can perform action a with probability p or action b with probability $1 - p$. The semantics is given in terms of probabilistic labelled transition systems in which the transition labels can be action types, probabilities, or both.

In [45] a classification of some probabilistic models is presented, distinguishing *reactive, generative* and *stratified* models. In a reactive system the probabilities of the transitions of an agent may depend on the environment in which the agent is placed: the choice of the action to be performed next is driven by what is offered externally. Once the environment has provided one action, the choice becomes internal and probabilistic. In each state in the transition diagram the

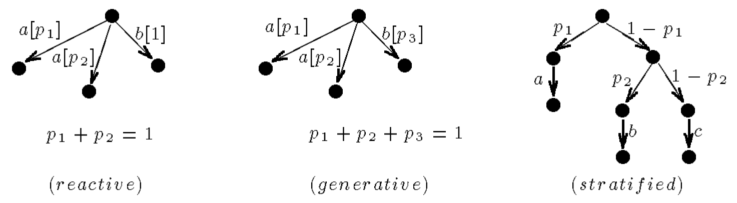


Figure 5.7: Different semantic models for probabilistic processes.

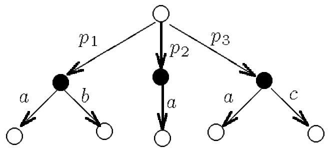


Figure 5.8: Alternating semantic model.

sum of the probabilities of outgoing transitions for each action type must be one (see the left part of Figure 5.7).

In a generative system the transition probabilities are independent of the environment and for each state the sum of the probabilities of the outgoing transitions must be (globally) one (see Figure 5.7 in the middle). If only a subset of actions are offered from the environment, new probabilities are computed by renormalisation.

Stratified models are a generalisation of generative models in which probabilistic and action transitions are kept separate. First a probabilistic choice is solved and then one action is performed. Notice that it is not possible to have nondeterministic transitions because after each probabilistic choice is performed, at most one action is enabled. The right part of Figure 5.7 shows an example of transition diagram for a stratified model.

In [20] a different approach is taken and a language that combines nondeterminism and probability is discussed. A probabilistic choice operator is proposed such that the designer can abstract away from the details of how choices are made but still provides information on the outcome of the choice itself. Probabilistic choice is independent from the environment: it is used for specifying an internal behaviour which cannot be influenced by synchronisation. The semantics is given in terms of labelled transition systems with different types of states, probabilistic and nondeterministic. Figure 5.8 shows an example of transition diagram: since there is a strict alternation between probabilistic states (white circles) and nondeterministic states (black dots) this semantic model has been called the *alternating* model.

Of course a key feature of these process algebras is again the equivalence relations that can be used to compare and analyse models which have been constructed. A probabilistic form of bisimulation has been proposed which aims to capture a notion of indistinguishability when it is assumed that the observer can witness the probability that an action occurs [32]. For this purpose, a probability measure, μ is defined over the transitions of a labelled transition system, $\mu : \mathcal{P} \times Act \times \mathcal{P} \longrightarrow [0,1]$ ($\mathcal{P}$ is the set of process terms). If we consider all the transitions into a set of process terms, via a given action, this can be extended to a probability measure $\nu : \mathcal{P} \times Act \times 2^{\mathcal{P}} \longrightarrow [0,1]$, such that

$$\nu(P \xrightarrow{a} S) = \sum_{P' \in S} \mu(P \xrightarrow{a} P').$$

The bisimulation for CCS is an equivalence relation and thus it generates equivalence classes over the set of all process terms $\mathcal{P}$. Exploiting this idea, a *probabilistic bisimulation* is defined to be an equivalence relation such that, for any two agents within an equivalence class, for any action $a \in Act$ and any equivalence class S, the probability measure ν of each of the agents performing an a action and resulting in an agent within S, is the same.

Definition 5.2 *A* **probabilistic bisimulation** $\mathcal{R}$*, is an equivalence relation over* $\mathcal{P}$ *such that whenever* $(P,Q) \in \mathcal{R}$ *then for all* $a \in Act \cup \tau$*, and for all* $S \in \mathcal{P}/\mathcal{R}$

$$\nu(P \xrightarrow{a} S) = \nu(Q \xrightarrow{a} S)$$

The definition of the probability measure μ, and consequently also ν, depends on whether the process algebra is reactive or generative. Larsen and Skou, [32] define $\mu(P \xrightarrow{a} P')$ for a reactive system, as the probability, given that P performs and action a, that P' is the derivative. In contrast, for a generative system, Jou and Smolka [47], define $\mu(P \xrightarrow{a} P')$ to be the probability that the transition $\xrightarrow{a} P'$ is the one that P performs.

Probabilistic bisimulation is a *strong* relation, in the sense that the silent action, τ, is treated in the same manner as any other action. Weak probabilistic bisimulation can be defined similarly to weak bisimulation for CCS [3].

5.4 Timed Process Algebras

The languages we have discussed so far (untimed and probabilistic) cannot express time delays between events: action executions take zero time and only relative ordering is represented via the traces of the processes. Over the years, various proposals for introducing time into process algebras have been made. These attempts follow two main lines: time can be either *deterministic* or *stochastic*. In the deterministic approach the possibility of finding a calculus for real time communicating systems in the style of CCS or CSP has been investigated [38, 2, 34, 53, 54, 36, 39]. The original motivation in the stochastic approach was that of combining system design and performance evaluation, starting from the first steps of the design [25].

The remaining part of this section is devoted to a brief description of some deterministic time extensions to CCS while languages which follow the stochastic approach will be described in more details in a following section.

Usually in deterministically timed PA some specific constructs are added to an untimed language and/or it is assumed that actions may be delayed. An explicit notion of time is defined by introducing a temporal domain T that can be either discrete (for example $\mathbb{N}$) or dense (for example $\mathbb{R}^+$). Systems are collections of cooperating components that may modify their states either by executing some actions or by letting time progress.

Probably the first timing extension to process algebra is Synchronous CCS [34]. In this language a discrete time domain is assumed and agents proceed in lockstep—i.e. at every instant each agent performs a single action. The expression $P = a.P'$, which in CCS represents an agent P which becomes P' after executing action a, has the following meaning in Synchronous CCS: *"the agent P, existing at time t, executes action a and then becomes P' at time $t+1$."*

In Temporal CCS [36] a discrete temporal domain is assumed and some operators are added to those of standard CCS. For instance, $(t).P$ represents the process that will evolve into P after exactly t units of time (*bounded* idling), $\delta.P$ represents the process that behaves as the process P, but is willing to wait any amount of time before actually proceeding (*unbounded* idling).

The language is given an operational semantics with two different types of transitions: action transitions, similar to those of standard CCS, which describe the functional aspects of the processes and time transitions which describe their temporal aspects. The underlying model is a transition system whose labels are either atomic actions or elements of an appropriate time domain.

In addition to delay operators which postpone the execution of an action by a given (or an unbounded) amount of time, other operators have been added to some timed PA to augment their modelling expressivity. For example, a *timeout* operator has been introduced. It uses two arguments P (the body) and Q (the exception) and a parameter t, and behaves as P if an initial action of P is performed within time t, otherwise it behaves as Q.

Yi's Timed CCS [54] is an extension of CCS which uses a set of idling actions defined as

$$\delta_T = \{\epsilon(t) \mid t \in T - \{0\}\}$$

Processes can evolve by executing CCS-like transitions or they can idle for a certain amount of time: $P \xrightarrow{\epsilon(t)} Q$ means that *"P will idle for t units of time and then will behave like Q."* The semantic rules of the language are those of CCS plus new ones that specify the idling behaviour of agents. A notion of strong bisimulation has been introduced to determine whether two agents are considered equivalent. This equivalence depends on the *capability* of the observer: an observer who cannot measure time will never tell the difference between two agents that execute their actions one in one second and the other in one year. This is a typical observer in CCS, but in Yi's Timed CCS a more powerful observer, who can also record the time delay between events, is required. In this context in fact, two agents are considered equivalent if they witness the same sequences of actions and the same sequences of delays. Notice that the duration of an activity is not associated with the action that represents it (which is executed in zero time) but it is modelled by a delay that precedes the action itself.

In Chen's Timed CCS [8], another timed extension of CCS, action prefix captures timing constraints which may apply to the action. The process $a(t)_e^{e'}.P$ can perform the action a between the times e and e' (inclusive); e and e' are called the lower bound and upper bound of action a respectively. For example an action a which can delay indefinitely is represented by $a(t)_0^{\top}$. Any occurrences of the time variable t which appear in the process P refer to the happening time of a. When the action is performed, say at time u, the variable t becomes bound to the value u. As well as a structured operational semantics, Chen gives a true concurrent semantics to his language in terms of timed synchronisation trees.

Some important properties which influence description capabilities have been defined for timed process algebras. Among them we mention *time determinism* and *action urgency*. The first property states that the progress of time should be deterministic: if a process P evolves into P' after a time t, and the same process evolves into P'' after the same amount of time, then P' and P'' must be the same process. Action urgency states that a process may block the progress of time and enforce the execution of an action before some delay (in some languages only the invisible action is urgent).

Another deterministic extension of PA has been proposed in [19]. In this language there is no distinction between the delay associated with an action and the action itself. The basic assumption is that actions are time-consuming and any action has an associated duration denoted by a natural number. Each sequential subsystem is equipped with a local clock that records the units of elapsed time due to the execution of actions which are local to the subsystem. When two subsystems interact through synchronisation they have to perform the same action at the same time. For this reason a sort of "busy waiting" is necessary when one subsystem is able to execute a synchronising action while the other is not. A notion of *performance equivalence* equates systems that perform the same actions with the same amount of time is discussed. Moreover the necessity of replacing deterministic time durations with time probabilistic distribution functions is recognised as a way to provide an uniform integration of the theories of process algebras and performance evaluation.

5.5 Stochastic Process Algebras

Recently the benefit of associating probabilistically distributed delays with the actions of a process algebra has been recognised, and introduced into several algebraic languages. Modelling the timing behaviour of systems by random variables rather than deterministic times allows the randomness of the real world to be captured. This is fundamental to performance evaluation. Incorporating this idea into a process algebra which facilitates compositional reasoning based on well-defined notions of equivalence, merges two previous distinct approaches to system representation. The result is a constructive methodology for the *specification* and *evaluation* of complex systems.

These new algebraic formalisms are known collectively as *stochastic* or *Marko-*

vian process algebras. Most of them adhere to Markovian assumptions, i.e. only random variables with a negative exponential distribution are used, but a few allow generally distributed random variables. However we will not consider these more general stochastic process algebras in this book, and will only discuss the Markovian process algebras. There are variations between the Markovian process algebras which have appeared in the literature but here we will concentrate on their common features. Therefore we will base our discussion on a *generic* language which we will simply call SPA.

SPA is based on an untimed process algebra in which the basic action has been extended with an exponentially distributed delay. Models in the language can be used to generate an underlying Markov process (see Chapter 9 which can then be used to derive performance measures for the modelled system.

5.5.1 Syntax and informal semantics

The basic process algebra of SPA is derived from CCS and CSP. In fact it is the language presented in Section 5.2.3. In this language systems are modelled as interactions of *components* that can perform a set of actions. Each action is given an associated random duration and is now termed an *activity.* An activity a is described by a pair (α, r) where α is the *type* of the activity and $r \in \mathbb{R}^+$ is the parameter of the negative exponential distribution governing its duration. Whenever a component P can perform an activity an instance of the given probability distribution is sampled. The resulting number specifies how long the component will take to *complete* the action.

The syntax of SPA, our abstract language, is defined as follows:

$$P ::= Nil \mid (\alpha, r).P \mid P + Q \mid P\|_S Q \mid P/L \mid A$$

The functional meaning of each operator is the same as in CCS and CSP (see Section 5.2), but we now also need to consider the durations of the activities.

- **Nil** As before, Nil represents the component which is not capable of performing any activities: a deadlocked component. In some timed process algebras such a component also cannot witness the passing of time and so results in the whole model being deadlocked. This is not the case in SPA—although this component cannot progress other components may be able to.

- **Prefix** This gives a component a designated first activity, i.e. the component $(\alpha, r).P$ performs the activity which has type α and a duration which is negative exponentially distributed with parameter r (mean $1/r$) and then evolves as P.

- **Choice** As in the untimed case, the component $P + Q$ represents a system which may behave either as component P or as Q. $P + Q$ enables all the current activities of P and Q and a *race condition* governs the resolution of the choice. This means that we may think of all the activities attempting to proceed but only the “fastest” succeeding. Thus the first activity to complete identifies one of the components which is selected as the component that continues to evolve; the other component is discarded. The outcome of the race will be random but dependent of the rates of the associated random variables. Thus the non-deterministic choice of CCS and CSP becomes probabilistic in SPA.

Whenever an activity completes the model evolves, now taking on the behaviour of the resulting component. Any other activity which was simultaneously enabled may remember the time for which it was enabled and start from that point whenever it is next enabled. Alternatively it may abandon its spent lifetime and start another lifetime whenever it is next enabled. Under the exponential assumption there is no difference between these two possibilities.

- **Parallel Composition** The component $P\|_S Q$ represents a system in which components P and Q work together to perform activities in the set S. The set S is called the *synchronising* or cooperation set. Both components proceed independently with any activities whose types do not occur in the set S. However, activities with action types in the set S are assumed to require the simultaneous involvement of both components. The resulting activity will have the same action type as the two contributing activities and a rate reflecting their rates. As we will discuss later, several possibilities exist for defining this resultant rate.
- **Hiding** As before, the component P/L behaves as P except that any activities of types within the set L are *hidden*, meaning that their type is not visible outside the component upon completion. Instead they appear as the unknown type τ and can be regarded as *internal delays* by the component. The timing characteristics of the hidden activities are unaffected.
- **Constant** Just as described for CCS, constants are components whose meaning is given by equations such as $A \overset{def}{=} P$. Here the constant A is given the behaviour of the component P. Constants can be used to describe infinite behaviours, via mutually recursive defining equations.

Sometimes a component will leave the rate of an activity *unspecified* (this is denoted $\top$, 0, or 1, depending on the language—here we will use $\top$). In this case we say that the component is *passive* with respect to that action type. Such a passive action must be shared with another component via synchronisation. In a final or complete model every passive action must be synchronised with at least one component active with respect to the action type and this component will determine the rate at which the shared activity occurs.

Analogously to GSPN, some of the stochastic process algebra languages also admit *immediate actions* which are completed in zero time with *priority* over timed actions (see Chapter 3). The implications of the inclusion of immediate actions are briefly discussed in Section 5.5.5.

5.5.2 Operational semantics

The deduction rules of the language operators, shown in Figure 5.9, outline the activities that a component can witness. Time is not represented explicitly, but it is assumed that timed activities take some time to complete and consequently the corresponding transitions represent some advance of time.

The rule for the prefix operator can be read as follows: in the component $(\alpha, r).P$ an activity of type α is enabled and the component behaves like P after its execution. The delay associated with the activity is exponentially distributed with rate r.

The other rules are straightforward and are presented without comment except for the rules concerning parallel composition. The first and the second rules represent the independent evolution of the two components while they are executing activities not in the cooperation set S. The third rule represents the cooperation between the two components to achieve a common task. In general, both components of the parallel composition will need to complete some work, as reflected by their own version of the activity, for the common activity to be completed. The rate R of the common activity is given by a function $\phi(r_1, r_2, P, Q)$ which reflects the rates of the individual activities. The different languages adopt different formulae for the definition of the new rate, all satisfying some algebraic requirements that are necessary in order to ensure that the equivalence notions which are defined are also congruences.

In PEPA [27] we have the following expression

$$\phi(r_1, r_2, P, Q) = \frac{r_1}{r_\alpha(P)} \frac{r_2}{r_\alpha(Q)} \min(r_\alpha(P), r_\alpha(Q))$$

where $r_\alpha(P)$ is the *apparent* rate of action type α in the P component and it is the rate at which an action type α appears to an external observer.

For any component P we can regard its total capacity for carrying out activities of a given type as the sum of the rates of the activities of that type which are enabled by P. This will be the apparent rate of the action type in P. We assume that when two components carry out α in cooperation their total capacity to complete the activity of that type is limited to the capacity of the slower component, i.e. the apparent rate of the synchronising activity is the minimum of the apparent rate of α in the two contributing components. Since each component may enable several α activities we assume that each independently chooses which activity instance takes part in the cooperation. Thus the rate of the synchronising activity is adjusted to reflect these probabilities, i.e. it is the product of the probability that each contributing activity is selected by its enabling component and of the apparent rate of the action type in the cooperation.

In MTIPP [22] the synchronisation between two or more processes leads to the observation of an action with a rate equal to the *product* of the rates of the individual processes i.e. $\phi(r_1, r_2, P, Q) = r_1 \cdot r_2$. We can distinguish two different situations: both actions are active or one action is active and the other is passive. In the first case (pairs of active actions) different *meanings* can be

Prefix

$$\frac{}{(\alpha, r).P \xrightarrow{(\alpha,r)} P}$$

Choice

$$\frac{P \xrightarrow{(\alpha,r)} P'}{P + Q \xrightarrow{(\alpha,r)} P'} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q'}{P + Q \xrightarrow{(\alpha,r)} Q'}$$

Parallel Composition

$$\frac{P \xrightarrow{(\alpha,r)} P'}{P\|_S\, Q \xrightarrow{(\alpha,r)} P'\|_S\, Q}\,(\alpha \notin S) \qquad \frac{Q \xrightarrow{(\alpha,r)} Q'}{P\|_S\, Q \xrightarrow{(\alpha,r)} P\|_S\, Q'}\,(\alpha \notin S)$$

$$\frac{P \xrightarrow{(\alpha,r_1)} P' \quad Q \xrightarrow{(\alpha,r_2)} Q'}{P\|_S\, Q \xrightarrow{(\alpha,R)} P'\|_S\, Q'}\,(\alpha \in S) \quad \text{where } R = \phi(r_1, r_2, P, Q)$$

Hiding

$$\frac{P \xrightarrow{(\alpha,r)} P'}{P/H \xrightarrow{(\alpha,r)} P'/H}\,(\alpha \notin H) \qquad \frac{P \xrightarrow{(\alpha,r)} P'}{P/H \xrightarrow{(\tau,r)} P'/H}\,(\alpha \in H)$$

Constant

$$\frac{P \xrightarrow{(\alpha,r)} P'}{A \xrightarrow{(\alpha,r)} P'}\,(A = P)$$

Figure 5.9: Operational semantics of SPA.

associated with the rates which can be interpreted as rates or as *scaling factors*. Let us consider two synchronising components representing a *Processor* and a *Task* executing an action α with rates r_1 and r_2 respectively. The rate r_1 may be interpreted as a measure of the processor speed while the rate r_2 may be interpreted as a scaling factor representing the dimension of the task, i.e. the number of work units composing it. By convention, a standard task has an associated rate equal to 1, values of $r_2 > 1$ represent "small" tasks and values of $r_2 < 1$ represent "big" tasks. Thus the resulting rate of the shared activity captures the amount of work to be done.

The synchronisation between passive and active agents is in general interpreted as a service provided by one agent and required by the other. In this case the rate of the passive action is equal to one (which is the neutral element with respect to the product) and therefore it is the active action that determines the global rate during the synchronisation.

In EMPA [5] synchronisation between processes requires that at most one active action is involved, while all the other actions must be passive. In this language passive actions have a rate equal to zero and the global rate of the synchronisation is given by the *maximum* of the individual rates. In this way the rate of the global action resulting from the synchronisation is determined by the rate of the active action only.

Notice that this choice has implications for compositionality since once an interaction involves one active participant only passive agents may be added later to the system.

In MPA [7] each activity is represented by a pair (α, r) where α, as usual, is the action type but r does not represent a rate; instead it describes the number of invocations of the action α. Normally an action is invoked only once by an agent, but r can also have other values, including real numbers. Moreover, it is assumed that each action α has associated a fixed exponential distribution with rate μ_α and that the invisible action τ has a rate μ_τ equal to one.

Each operation (action) needs a basic time to be performed. If an agent is faster, the parameter r associated with the action is greater than one, if it is slower, the parameter r is smaller than one. Thus, in a certain sense, r can be related to the speed of the agent.

Since each action has a fixed rate the interaction of two α actions results in an action with the same rate μ_α. However, the number of invocations of action α equals the *product* of the number of invocations of both involved activities. In this way, each invocation in one agent is combined with all the invocations in the other agent.

Transition diagram/Derivation graph We have seen in the previous sections that PA terms can be mapped onto transition diagrams whose arcs are labelled with action names, probabilities, and elements of an appropriate time domain.

In SPA actions are time consuming and we can derive a transition diagram whose arcs are labelled with pairs consisting of the action type and the corresponding rate. However, we must take some care. Consider a simple agent P which will repeatedly carry out the action (α, r). For a classical process algebra (and for qualitative analysis) we need only consider which actions are possible in an agent. Thus the agent $P+P$ has the same behaviour as the agent P — both are capable of an action α and subsequently behave as P — so these agents are considered equivalent (see the equational law in Section 5.2). In SPA multiple instances of an action become apparent because the duration of an action of that type will appear to be the minimum of the corresponding random variables (race policy). In the case of exponentially distributed durations this means that the global rate of an action will be the sum of the rates. Thus $P+P$ appears to carry out the first α action at twice the rate of the agent P. Consequently P and $P+P$ cannot be regarded as equivalent. As a consequence in the semantics of the language the number of instances of a transition between states, together with their rates, has to be recognised.

The underlying Markov process There is a clear correspondence between the labelled transition system of an SPA model and a continuous time Markov process. The Markov process underlying any finite SPA component can be obtained directly from the transition diagram: a state of the Markov process is associated with each node of the diagram and the transitions between states are defined by the arcs of the diagram. 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 transition diagram.

5.5.3 The PLC multicomputer example in SPA

Our PLC example can be specified with SPA by extending the previous untimed specification associating a rate with each action. We obtain the following description:

$$\begin{array}{lcl}
Comp_1 & = & (start\_cycle, r_1).Comp_2 \\
Comp_2 & = & (local\_comp, r_2).Comp_3 \\
Comp_3 & = & (end\_cycle, r_3).Comp_1 + (req\_ext\_data, r_4).Comp_4 \\
Comp_4 & = & (acq\_bus, r_5).Comp_5 \\
Comp_5 & = & (read\_ext\_data, r_6).Comp_2 \\
 & & \\
Bus_1 & = & (acq\_bus, \top).Bus_2 \\
Bus_2 & = & (read\_ext\_data, \top).Bus_1 \\
 & & \\
PLC & = & (Comp_1 \|_{\{start\_cycle\}} Comp_1) \|_{\{acq\_bus, read\_ext\_data\}} Bus
\end{array}$$

The rates of the actions acq_bus and $read\_ext\_data$ are left unspecified in Bus_1, meaning that this component is passive with respect to them.

5.5.4 Equivalence relations in SPA

Equivalence relations have been used in untimed, probabilistic and timed PA to compare components and to replace a component by another which exhibits an equivalent behaviour, but has a simpler representation. This technique still applies in SPA where different notions of equivalence that cover the functional aspects, the temporal aspects and both of them, have been defined.

As previously, we need to specify the capabilities of the observer. Can the observer record the rate of each activity? Can he record the relative frequency with which alternative activities occur in a given component? Usually it is assumed that the observer has no memory of the past history of the components and bases his comparison only on the current behaviour.

The definition of equivalences for SPA should conservatively extend the notion of bisimilarity introduced in Section 5.2, since this has proved to be fundamental for PA. The difficulty is that now we have to consider not only qualities but also *quantities*, i.e. the rates associated with the action types. In contrast, bisimilarity only considers a (logical) quality: either there is a move between two states or it is impossible.

A fundamental equivalence notion for SPA, called *strong equivalence* [26] or *Markovian bisimulation* [22], considers both qualities and quantities and it has been introduced to ensure indistinguishability under experimentation. We briefly introduce this equivalence here and the reader is invited to see Chapter 16 for further details.

Some definitions are necessary. If P can, by some action, evolve to become Q then Q is said to be a *derivative* of P. The *transition rate* between two components P and Q is denoted by $q(P,Q)$ and is the sum of the activity rates labelling arcs connecting node P to node Q in the transition diagram. The *conditional transition rate* from P to Q via an action type α is denoted by $q(P,Q,\alpha)$. This is the sum of the activity rates labelling arcs connecting the corresponding nodes in the transition diagram which are also labelled by the action type α. The conditional transition rate is thus the rate at which a system behaving as component P evolves to behaving as component Q as the result of completing an activity of type α.

If we consider a *set* of possible derivatives S, the *total conditional transition rate* from P to S, denoted $q[P,S,\alpha]$, is equal to the sum of the conditional transition rates from P to components Q_i belonging to S:

$$q[P,S,\alpha] = \sum_{Q_i \in S} q(P,Q_i,\alpha)$$

The concept of total conditional transition rate is the basis for the definition of strong equivalence since two components are considered strongly equivalent if for any action type α, the total conditional transition rates from those components to any equivalence class, via activities of this type, are the same.

Definition 5.3 *A binary relation $\mathcal{R}$ over components is a* **strong equivalence** *if whenever $(P,Q) \in \mathcal{R}$ then for all α and for all equivalence classes S induced* *by* $\mathcal{R}$,

$$q[P,S,\alpha] = q[Q,S,\alpha]$$

In this definition the total conditional transition rate is used analogously to the probability measure ν in Definition 5.2 and [32].

We illustrate the notion of strong equivalence by considering again the PLC example already discussed in Sections 5.2.3 and 5.5.3. Let us take the term $Comp_1\|_{\{start\_cycle\}}Comp_1$ consisting of two computers synchronising on $start\_cycle$, without considering any interaction with the global bus. It is possible to show that $Comp_1\|_{\{start\_cycle\}}Comp_1$ is strongly equivalent to the term written below:

$$\begin{array}{rcl}
Comp^{eq} &=& (start\_cycle, r_1).Comp_1^{eq} \\
Comp_1^{eq} &=& (local\_comp, 2r_2).Comp_2^{eq} \\
Comp_2^{eq} &=& (end\_cycle, r_3).Comp_3^{eq} + (local\_comp, r_2).Comp_4^{eq} \\
&& \qquad\qquad +(req\_ext\_data, r_4).Comp_5^{eq} \\
Comp_3^{eq} &=& (local\_comp, r_2).Comp_6^{eq} \\
Comp_4^{eq} &=& (end\_cycle, 2r_3).Comp_6^{eq} + (req\_ext\_data, 2r_4).Comp_7^{eq} \\
Comp_5^{eq} &=& (local\_comp, r_2).Comp_7^{eq} + (acq\_bus, r_5).Comp_8^{eq} \\
Comp_6^{eq} &=& (end\_cycle, r_3).Comp^{eq} + (req\_ext\_data, r_4).Comp_9^{eq} \\
Comp_7^{eq} &=& (end\_cycle, r_3).Comp_9^{eq} + (req\_ext\_data, r_4).Comp_{10}^{eq} \\
&& \qquad\qquad +(acq\_bus, r_5).Comp_{11}^{eq} \\
Comp_8^{eq} &=& (read\_ext\_data, r_6).Comp_3^{eq} + (local\_comp, r_2).Comp_{11}^{eq} \\
Comp_9^{eq} &=& (acq\_bus, r_5).Comp_{12}^{eq} \\
Comp_{10}^{eq} &=& (acq\_bus, r_5).Comp_{13}^{eq} \\
Comp_{11}^{eq} &=& (read\_ext\_data, r_6).Comp_6^{eq} + (end\_cycle, r_3).Comp_{12}^{eq} \\
&& \qquad\qquad +(req\_ext\_data, r_4).Comp_{13}^{eq} \\
Comp_{12}^{eq} &=& (read\_ext\_data, r_6).Comp^{eq} \\
Comp_{13}^{eq} &=& (read\_ext\_data, r_6).Comp_9^{eq} + (acq\_bus, r_5).Comp_{14}^{eq} \\
Comp_{14}^{eq} &=& (read\_ext\_data, r_6).Comp_{12}^{eq}
\end{array}$$

Essentially we have taken advantage of the symmetry between the two computers which was discussed in Section 5.2.6, and formed a single component which preserves their combined behaviour from the point of view of an external observer. The transition diagrams underlying $Comp_1\|_{\{start\_cycle\}}Comp_1$ and $Comp^{eq}$ have 25 states and 51 transitions, and 15 states and 26 transitions, respectively. Portions of the corresponding transition diagrams are shown in Figures 5.10 and 5.11 where for readability we have omitted the states names and we have associated the rates only with some arcs labels.

These two terms are strongly equivalent but the second has a smaller state space. Now we can take advantage of the fact that strong equivalence is a congruence and we can substitute $Comp^{eq}$ for $Comp_1\|_{\{start\_cycle\}}Comp_1$ in a more complex system. For example we can transform

$$PLC' = \; (Comp_1\|_{\{start\_cycle\}}Comp_1\|_{\{start\_cycle\}}Comp_1\|_{\{start\_cycle\}}Comp_1) \\ \|_{\{acq\_bus, read\_ext\_data\}}Bus_1$$

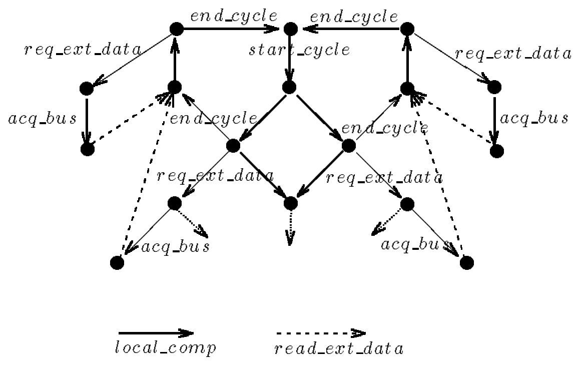


Figure 5.10: Portion of the transition diagram of $Comp_1 \|_{\{start\_cycle\}} Comp_1$.

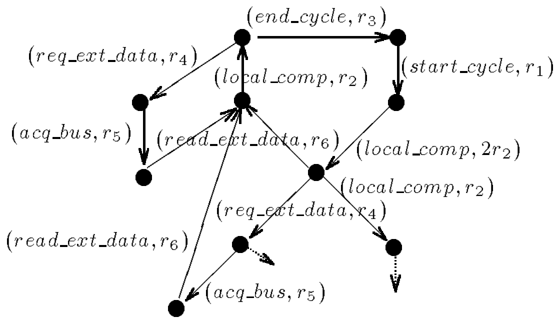


Figure 5.11: Portion of the transition diagram of $Comp^{eq}$.

into

$$PLC^{eq} = (Comp^{eq} \|_{\{start\_cycle\}} Comp^{eq}) \|_{\{acq\_bus, read\_ext\_data\}} Bus_1$$

being sure that they have an equivalent behaviour. The advantage is that the transition diagram of PLC^{eq} has 180 states and 557 transitions while in the case of PLC' we obtain 512 states and 1857 transitions

This notion of equivalence can be used to compare and replace components by simpler ones but it also has interesting stochastic properties which can be exploited for model aggregation and for the efficient analysis of SPA models. These features will be discussed in Chapter 16.

5.5.5 Immediate actions

Some SPA languages allow the use of *immediate* actions which are executed in zero time with priority over timed actions. Immediate actions have been introduced in EMPA [5] taking inspiration from the work on GSPN. Exponentially timed actions have an associated rate $\lambda \in \mathbb{R}^+$ while immediate actions have a rate $\top_{l,w}$ where $l \in \mathbb{N}^+$ is the priority level and $w \in \mathbb{R}^+$ is the associated weight. Different types of choices can be expressed within this language. When the choice involves exponentially timed actions only, the race policy determines which action will be performed. In the case of a state enabling both timed and immediate actions, the immediate actions with the highest priority level are the only actions actually executable, the choice being performed on the basis of their associated weights.

A different approach has been followed in [23] where the basic MTIPP language has been enriched with immediate actions which are represented by their names only. These actions can be *external* or *internal* (denoted τ) referring to whether the environment can influence their execution.

Again, a crucial aspect of the language is the choice operator since the behaviour of a term like $P + Q$ depends on the nature of the choice alternatives. If the choice involves Markovian actions only, as in Figure 5.12 (left), the race policy is used to determine the future behaviour of the term. If the choice involves immediate actions only, the decision is taken on the basis of what is offered from the environment; when the environment offers several of the actions enabled in P and Q, the choice becomes nondeterministic (see middle of Figure 5.12, under the hypothesis that both a and b are offered).

Finally, if immediate and Markovian actions appear in a choice, immediate actions *might* happen instantaneously if they are not blocked from the environment. In particular, since internal actions cannot be prevented, they always happen with priority over timed actions as shown in the right part of Figure 5.12.

Notice that in this language the hiding operator plays a special role since it can be used to give priority to immediate external actions. In the expression $P = (\alpha, r).P_1 + b.P_2$ the action b is executed instantaneously only if the environment also offers the same action instantaneously; otherwise, the environment governs when action b may happen. The environment may also determine that action

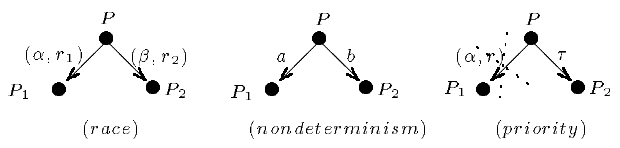


Figure 5.12: Different types of choice: race, nondeterminism, priority.

α occurs first. However, by hiding action b, we can be sure that it cannot be blocked and therefore will always (invisibly) prevent the occurrence of action α. In other words, a priority has been associated with action b by means of hiding.

Reduction of the Nondeterminism In Section 3.2 we discussed how the definition of a TPN starts from a *nondeterministic*, untimed PN model. The inclusion of time is done in such a way that the amount of nondeterminism is reduced, eventually ensuring that the behaviour of the system is specified precisely enough to derive performance measures. In the PN setting, different ways to attack the problem of reducing nondeterminism have been identified. Two of them re-appear in the context of SPA. The incorporation of exponentially distributed delays into an untimed process algebra leads to a stochastic reduction of nondeterminism in the sense of Section 3.2.2. Due to the race condition, nondeterminism is replaced by assigning a probability to each alternative.

In the presence of immediate actions the situation is different: nondeterminism between timed transitions is reduced stochastically, while nondeterminism between immediate transitions is not. This implies that the amount of nondeterminism can make the specification too imprecise to allow the derivation of performance measures. To enhance the precision of such specifications one relies on a variant of nondeterminism reduction in the style of Section 3.2.1. A reduction is performed by postulating some *external influence*. As an example, consider the middle example in Figure refchoice. The term $a.P_1 + b.P_2$ describes a nondeterministic choice, when (and only when) both action a and action b may happen. By postulating an additional external influence we may resolve this choice. For instance, a synchronisation on action b with the inactive process Nil prevents the occurrence of the b branch of the choice. In this way, the amount of nondeterminism is reduced.

Deterministic reduction in the style of Section 3.2.3 has also received some attention in the more general context of models with nondeterminism and probabilities [51]. For this purpose the notion of a *scheduler* is introduced. A scheduler is a function that for any state determines the next transition to be executed, taking into account the history of the system. If the scheduler has the possibility to "roll dice", this form of deterministic reduction turns into a stochastic reduction.

5.6 Comparison with Petri nets

As we have seen throughout this chapter GSPN and SPA have undergone similar developments. In both cases the original definitions did not include any temporal information, and they were used only for the qualitative analysis of concurrent systems. The stochastically timed extensions allowed the study of the systems' quantitative properties too, since the two formalisms now formed high-level description languages for Markov processes. In both cases also, as well as the stochastically timed actions there have been subsequent developments to include immediate actions.

In this section we consider similarities and differences between the two formalisms. We start with the definition of a mapping between them which provides a formal means of investigating the relationship between them. We show how to map the basic operators of our SPA language into GSPN models. In the remainder of the section we compare the formalisms in terms of model construction, qualitative and quantitative analysis. Our comparison is largely informal as we focus on the facilities that are offered to the modeller, rather than theoretical definitions of modelling power etc.

Many comparisons of untimed Petri nets and process algebras have appeared in the literature, concentrating on their different representations of causality and concurrency. Our motivation is different: our intention is to help the reader develop a better understanding of the formalisms, individually and in relation to each other.

5.6.1 Mapping SPA operators into GSPN models

We have already seen in Section 5.5.2 that operational rules are associated with each algebraic operator of SPA, to provide a structured operational semantics. Now we take a different approach to semantics and we associate a net model with each algebraic operator. We follow the *compositional* approach already proposed in [18, 48] for untimed models, and in [4, 46] for timed models.

This compositional approach to (net) model construction requires that the net of a complex language expression is obtained starting from the subnets of the composite subterms. For each operator *op* in the language, we will describe the operation $op_{\mathcal{N}}$ that has to be performed at the net level to reproduce its effect. These operations are based on the well-known net operations of transition and place superposition.

We consider *labelled* GSPN defined as 7-tuples $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{Post}, \mathbf{w}, \mathbf{m_0}, l\rangle$ where $P, T, \mathbf{Pre}, \mathbf{Post}, \mathbf{w}, \mathbf{m_0}$ are the same of the previous chapters and l is the labelling function $l : T \rightarrow Act$ that maps transition names into action names, whose meaning will be immediately clear in the translation examples below. We are not introducing a formal definition here, but rather explaining everything in terms of simple examples; the reader is invited to consult [46] for further details.

The most straightforward mapping is that of the Nil operator which can be translated into a single marked place, not connected to any transition.

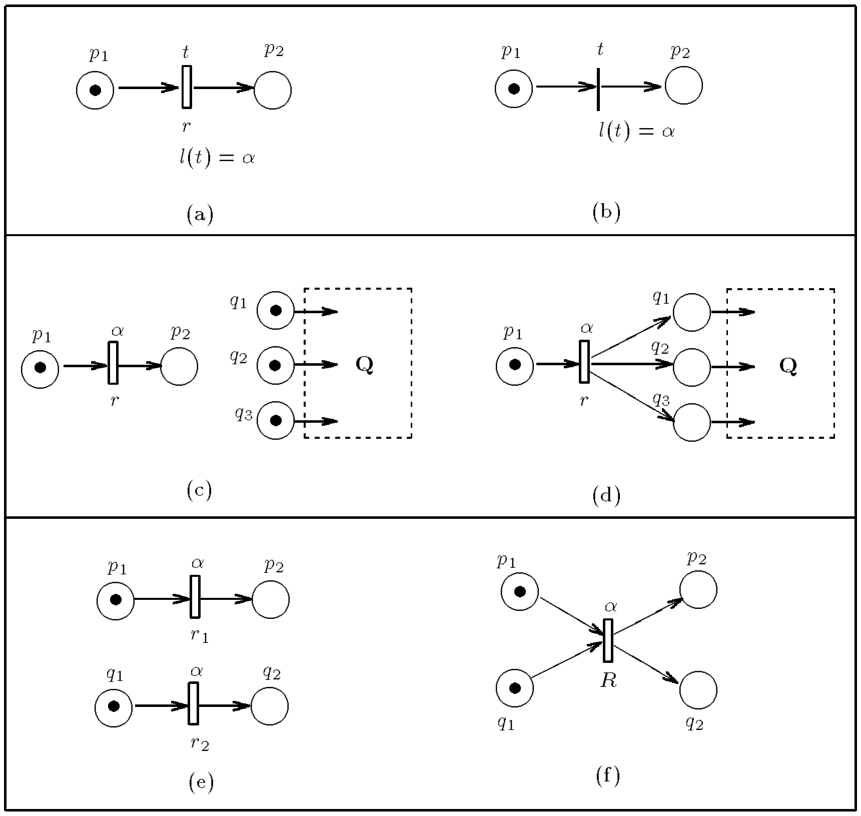


Figure 5.13: GSPN mapping of SPA operators.

The (timed) basic element in our SPA is the activity (α, r) composed of two pieces of information: the action type α and the rate r. Its translation is shown in Figure 5.13(a): p_1 and p_2 are the input (initial) and the output (final) places of the basic element and the timed transition t models action α. In the case of immediate actions we have the same net structure, but the transition t is now immediate, as shown in Figure 5.13(b).

Note the difference between *transitions* and *actions*: since in SPA different terms can perform actions of the same type, several transitions may represent the same action type. However, by definition, transitions need to be distinguishable, i.e., they must have distinct names. Therefore, we use the labelling function l to map transition names into action names[1] so that it is possible that $l(t_i) = l(t_j) = \alpha$ but $t_i \neq t_j$.

Consider the term $P = (\alpha, r).Q$, constructed by the prefix operator. Its mapping into a net model is obtained by first translating the components (α, r) and Q in isolation (see Figure 5.13(c)); then, to represent the "$\cdot$" operator, the

[1]In the following figures we associate the action name $l(t)$ with each transition t.

final place of (α, r) is superposed on all the places which are initially marked in the net modelling Q (see Figure 5.13(d)). Notice that the initial places of Q (q_1, q_2, q_3) are no longer marked.

The translation of the term $P \|_S Q$ requires superposition of transitions and is again performed in two steps. First we map P and Q into GSPN models; then we use an appropriate scheme to translate the $\|_S$ operator. We distinguish two cases:

action types $\alpha \notin S$**:** we consider the union of the nets representing P and Q[2];

action types $\alpha \in S$**:** we superpose transitions with the same label α and compute appropriate synchronisation rates.

The net depicted in Figure 5.13(f) shows the translation of the term $P = (\alpha, r_1).P_1 \|_{\{\alpha\}} (\alpha, r_2).Q_1$. Both processes execute an α activity that belongs to the synchronising set; their parallel composition is obtained by superposing the two α transitions. The value of the rate R depends from the choice of the function $\phi(r_1, r_2, P_1, Q_1)$ discussed in Section 5.5.2.

It is only possible to synchronise actions of the same type, i.e. timed with timed, immediate with immediate. This means that at the net level we will (correctly) superpose only transitions of the same type.

The translation of choice uses place superposition. As previously, for the term $P + Q$ we first translate the components P and Q, and then we translate the operator "+" using the Cartesian product of their input places. For example, consider $R = P + Q$ where $P = (\alpha, r_1).P_1 \| (\beta, r_2).P_2$ and $Q = (\gamma, r_3).Q_1$.

The nets representing the two components, P and Q, and the one representing R are shown in Figure 5.14(a) and Figure 5.14(b). The initial places in the composed net are obtained by considering the Cartesian product of the input places of the nets of P and Q (see the places p_1, p_2 and q_1 and the places $p_1 \times q_1$ and $p_2 \times q_1$ in the figure).

The translation of the term P/H is obtained by mapping the term P into the corresponding GSPN model, and by relabelling all the transitions whose associated labels belong to the set H to τ.

In our SPA there is no explicit recursion operator and infinite behaviours are obtained using constants. For example, the expression $P \overset{def}{=} (\alpha, r_1).(\beta, r_2).P$ describes a component that can perform an infinite number of α and β actions. The translation of recursive terms like P can again be performed in two steps. First, we translate P as shown in Figure 5.14(c); then we recognise that we have to repeat the same behaviour. Instead of repeating the same net structure, we can close the net obtaining a GSPN model in which the transitions labelled α and β can fire infinitely many times (Figure 5.14(d)). Here we have restricted ourselves to a simple form of recursion in which the recursive term is composed of a sequence of prefixing operators. In [18] a more general translation for the CCS recursion is discussed and several problems which arise when translating recursive terms are presented; details can be found in the literature.

[2]When the set S is empty, we have the (pure) parallel composition of independent components which is translated as the union of the starting nets (Figure 5.13(e)).

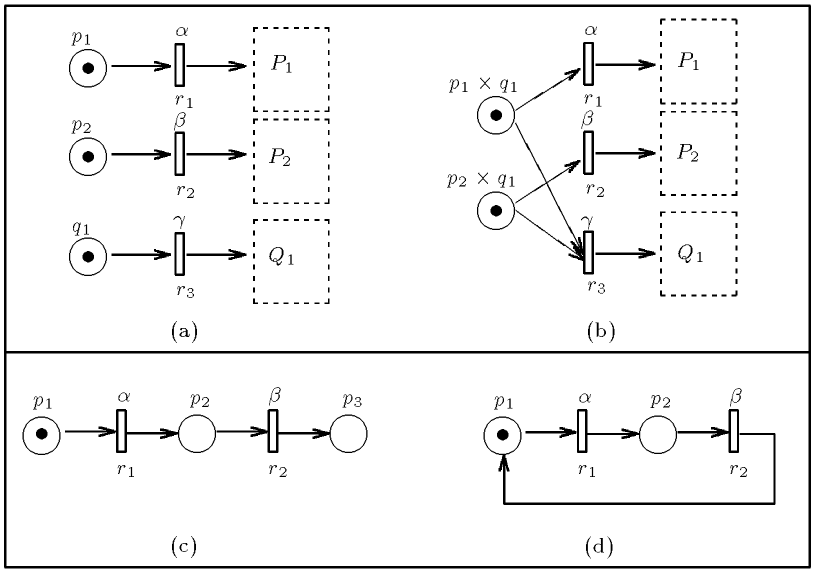


Figure 5.14: GSPN mapping of SPA operators (continue).

We end this section by inviting the reader to observe that, by applying the semantics rules just presented to the PLC model of Section 5.2.3, we obtain a net which is isomorphic[3] to that shown in Figure 2.9 of Section 2.3.1.

5.6.2 Comparison of model construction styles

In this section we examine the different styles in which GSPN and SPA, as high-level description languages for Markov processes, express the behaviour of systems.

At a notational level the difference seems significant since SPA is a textual language and GSPN has a graphical notation. However, at the other extreme, if we consider the class of Markov processes which can be expressed, it is clear that *any* Markov process can be expressed as a degenerate form using either formalism. In GSPN a place is associated with each state and appropriate transitions are inserted between the places to represent the transitions in the Markov process. The place corresponding to the initial state is marked by a single token. Similarly for SPA: a component is associated with the initial state of the Markov process, with a derivative for each subsequent state; activities capture the transitions. Neither of these comparisons reflect the modelling styles of the languages, which is what we wish to compare: for example, in the degenerate GSPN the notions of distributed state and local evolution are lost.

Five different aspects of modelling style, which highlight the differences between the formalisms, are considered below.

[3]Ignoring timing information.

State vs action orientation GSPNs have a very clear notion of state, the distribution of tokens in the places of the net. We can regard a GSPN model as an association of a state (the initial marking) with a graph structure, where the graph structure specifies how the state is modified (state evolution). Note that there are two distinct languages: one to define the structure, and one to record the state.

There is no explicit notion of state in the syntax of SPA. However, at the semantic level, in the labelled transition system, a state is associated with each syntactic term. Thus for an SPA model each derivative of the initial expression representing the model is considered to be a state.

Observe that, in general, it is not possible to distinguish by notation between the derivatives of a model (its states) and the complete model specification—they are all just language terms. The model definition will usually consist of a set of equations: at least one for each component and at least one to specify the interactions between the components. This last equation defines the structure of the model, a structure which will be reflected in all the derivatives, or states. So in some senses this is analogous with the graph structure of the GSPN. But note that it is also analogous to the initial marking since this equation specifies how many instances of each component are active in the model.

From a modelling point of view, GSPN is focussed on both states and actions, while SPA is focussed on actions. Given an arbitrary marking of a GSPN model it is usually possible, by considering the number of tokens in a given place, to immediately infer the state of the system, e.g. "bus is free." In contrast, the information that can be immediately extracted from an SPA model during its evolution is in terms of the actions which the model (system) could perform, e.g. "acq_bus" action is enabled. This may implicitly tell us that the bus is currently free. This distinction has consequences for the definition of performance measures which will be discussed in Chapter 8.

To see how the generation of states in the two paradigms differs, consider a simple system in which a job has a choice between two possible evolutions: it may perform action α at rate r_1 followed by action β at rate r_3, or it may choose to perform action α at rate r_2 followed by action β at rate r_3. A representation of this system in the two formalisms is given in Figure 5.15, by SPA on the left and GSPN[4] on the right. The derivative set of the SPA model has two elements, $\{P_1 + P_2,\ (\beta, r_3).(P_1 + P_2)\}$, while the reachability set of the GSPN model has three different states, $\{(1,0,0),\ (0,1,0),\ (0,0,1)\}$.

In SPA, after action (α, r_1) $((\alpha, r_2))$ takes place, action (β, r_3) is executed, without making any distinction of whether β is executed as an action of the first component (P_1) or of the second one (P_2). In the GSPN model instead the two β actions are represented by the two distinct transitions t_3 and t_4 triggered by a condition on two different places. Thus there is a "history" of which component executed the α action remembered by the model.

This example shows how, in SPA, terms with a common future are considered to represent the same state, whereas in GSPNs this is not necessarily so. We

[4]This GSPN model is obtained by applying the semantic rules introduced in Section 5.6.1.

$$\begin{aligned} Q &\stackrel{def}{=} P_1 + P_2 \\ P_1 &\stackrel{def}{=} (\alpha, r_1).(\beta, r_3).Q \\ P_2 &\stackrel{def}{=} (\alpha, r_2).(\beta, r_3).Q \end{aligned}$$

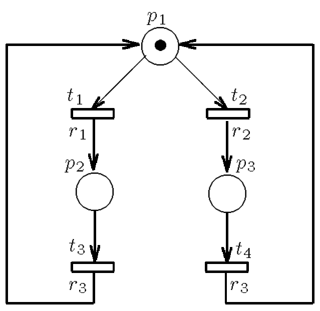


Figure 5.15: SPA and GSPN models of a simple system.

could have modelled the system with a GSPN where p_2 and p_3 (t_3 and t_4) are fused in a single place (transition), but this does not appear a very "natural" model for the given system, since the two possible evolutions of the job are described separately. Notice that this folding would be possible only because both transitions t_3 and t_4 have the same rate. Nevertheless, when the two β actions (the two transitions t_3 and t_4) have different rates, the two corresponding states are kept separate in both formalisms.

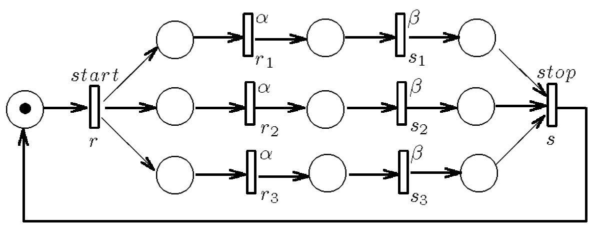
Static vs dynamic model entities Process algebras distinguish between dynamic and static combinators. Choice is dynamic since after a choice has been made the syntactic form of the component will, in general, be different: e.g. $P{+}Q \longrightarrow P'$. Parallel composition is a static combinator since the syntactic form of the component is the same after evolution regardless of whether a shared or an individual activity was completed, e.g. $P\|_S Q \longrightarrow P'\|_S Q$.

In SPA, entities within the system are represented as components in the model. If a model is to generate an ergodic Markov process it is necessary that the parallel components of a model are static—they are not created or destroyed as the model evolves. The initial term of an SPA model shows all the parallel components which are going to exist during the life of the model, and therefore an ergodic SPA model will have the same number of such terms throughout its evolution. Choices may occur within such components and these represent alternative modes of behaviour but not alternative structures.

In GSPN there is no notion of static components. However, a related concept is that of *P-semiflows* of the net (see Chapter 6). These are subsets of places such that a weighted sum of tokens in those places is constant for all reachable states. We observe that if an SPA term is built as the composition of N distinct components, then the equivalent GSPN has at least N P-semiflows where all the places have weight equal to 1. We refer to Chapter 6 for the discussion about P- and *T-semiflows*.

In a GSPN model, entities within the system are associated with the tokens of the Petri net. As the net evolves the number of tokens may vary, reflecting the dynamic behaviour of the system, according to the firing rule. This models the interactions between the entities in the system. For example, in the case of the GSPN model of the PLC example, certain tokens represent the computers and others the buses. Tokens representing computers move from state to state (e.g. performing a local computation, reading external data) through the firing of transitions; when a computer is using the bus this is represented as a single token, although two entities are involved.

As another example, consider a simple parallel program containing a parbegin/parend section. In the GSPN this can be simply modelled using a fork & join structure as shown in the upper part of Figure 5.16. Representing this system in SPA produces an alternative view of the system (see the lower part of Figure 5.16). The GSPN representation has one process which becomes split into three and then recombined to form a single process again. The static nature of the SPA models forces a representation with three processes which are initially and finally constrained to act together, only free to act independently during the middle phase of their execution.



$$\begin{aligned}
P_{i0} &\overset{def}{=} (start, r).P_{i1}\\
P_{i1} &\overset{def}{=} (\alpha, r_i).P_{i2}\\
P_{i2} &\overset{def}{=} (\beta, s_i).P_{i3}\\
P_{i3} &\overset{def}{=} (stop, s).P_{i0}\\
System &\overset{def}{=} ((P_{10} \|_{\{start,stop\}} P_{20}) \|_{\{start,stop\}} P_{30})
\end{aligned}$$

Figure 5.16: GSPN and SPA models of the fork & join structure.

Modelling abstraction One of the skills of an experienced modeller is choosing an appropriate level of abstraction at which to construct a model. Although this is largely a question of judgement it is aided by flexibility in the way a system may be presented in a paradigm.

For example, consider a system which is comprised of two identical instances of an entity, which are independent. In SPA this would be modelled as the component $Q \overset{def}{=} P \| P$. The component P could have any behaviour but for simplicity we assume that it repeatedly carries out activity (α, r) followed by (β, s): $P \overset{def}{=} (\alpha, r).(\beta, s).P$.

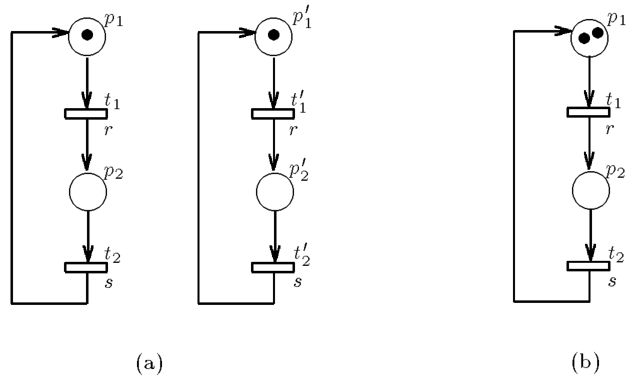


Figure 5.17: Alternative GSPN representations of two instances of the same entity.

There are two possible GSPN models of this behaviour. In the first, the net representing the behaviour of the entity is constructed, and repeated instances of the component are represented by repeated instances of the same net (Figure 5.17(a)). Alternatively, the single net structure representing the entity may be marked by two tokens in its initial place to represent the repeated structure (Figure 5.17(b)), if we assume an infinite server discipline for the transition t_1.

In SPA repeated instances of the same entity are usually distinguished so all possible interleavings of states are represented, i.e. we distinguish between $P' \| P$ and $P \| P'$. However, recent work [17, 24] has aimed to take advantage of the fact that the identity of the component which has completed the activity (α, r) is unimportant. This is analogous to the compact GSPN representation of Figure 5.17(b) in which the marking $(1, 1)$ models a such a situation.

Compositionality Compositionality is a central feature of model construction in SPA, resulting in models which are easy to understand and readily modified. The expression $Q \stackrel{def}{=} P \| P$ shows that the system is comprised of two identical but independent components, without detailed information about the behaviour of P. It follows that an SPA term may have a lot of embedded behaviour, defined in a separate expression. This leads to a hierarchical approach to model construction. The resulting model has a structure which reflects the structure of the system itself. Moreover this structure may be exploited during analysis. Model components can be developed by different modellers and libraries of re-usable components may be established.

In contrast, with GSPN the model is constructed as a *flat* representation of the system, all modelling primitives conveying the same amount of information. In fact, both the notions of state and firing can be viewed as compositional since their are based on the knowledge of local information. Hence there is nothing within the GSPN formalism which makes compositionality impossible. Indeed, although Petri nets (timed or untimed) are not compositional in nature, there have been efforts to add composition operators to the formalism: composition is based on superposition of places and transitions (see Chapter 12), and more recently it has been defined in the more structured approach of the Box Calculus [12], and in the case of Well Formed nets in [37]. Chapter 16 will discuss in detail the notion of compositionality at the net level.

Operator abstraction We can consider two forms of operator abstraction in performance modelling paradigms: *functional* and *temporal* abstraction. Functional abstraction allows some behaviour of the system to be abstracted away because it is more detailed than necessary for the current model. Temporal abstraction allows some of the timing information within a system to be abstracted away as irrelevant in the current model.

Temporal abstraction is provided by immediate transitions or actions when they are used to represent timed actions, the duration of which is negligible compared with that of other actions within the model. It is assumed that these events do not have a significant impact on the performance of the system. Note that, at a conceptual level, this is distinct from the use of immediate transitions to represent logical actions, to which no time can be associated.

Functional abstraction in SPA is provided by the hiding operator. Activities of type τ are considered to be internal to the component in which they occur. Thus hiding allows an interface to a model or component to be defined as we have already discussed. This is particularly powerful when used in conjunction with parallel composition as it may restrict the interactions of a component. We invite the reader to see Chapter 16 for a discussion on functional abstraction in the context of (compositional) SPN.

5.6.3 Qualitative and quantitative analysis

Since GSPN and SPA have both evolved from formal system description techniques, models in either paradigm can be regarded as a functional representation of the system, as well as a performance representation. The choice, in both GSPN and SPA, to use a distribution with infinite support for the delays, allows functional properties of the timed models to be proved with the same techniques used for their untimed counterpart.

Graph-based analysis, also called state space analysis, may be used to answer many questions about system behaviour when applied to the reachability or derivation graph. For example, the presence of a *dead* state, i.e. a node with no output arc, can be checked on the graph, as well as the reachability of a given state M' starting from a state M can be checked by looking for a direct path in the graph connecting the corresponding nodes.

State space analysis techniques are very powerful, since they allow the proof of many properties of interest by inspection of the graph which contains all possible evolutions of the model. In general, they are considered to be very expensive because the space and time complexity of the graph construction algorithm can exceed acceptable limits. However, in the case of GSPN and SPA models, intended for performance evaluation, construction of this graph is essential in any case to generate the underlying Markov process. State space analysis techniques as well as other analysis techniques are discussed in Chapter 6.

The solution techniques applied to compute quantitative results in GSPN and SPA are identical, based on the numerical solution of the underlying Markov process. Starting from a GSPN (SPA) model, the associated reachability (derivation) graph is obtained and then reduced to the corresponding continuous time Markov process. This is then numerically solved to compute the steady state probabilities.

Being a more mature formalism, there has been more work on efficient algorithms for finding and solving this Markov process in the case of GSPN, and there has been a certain effort towards "less expensive" solution methods. Some of the efficient algorithms for the construction and solution of the associated Markov process have already been imported into SPA, and work is progressing on others. Again, we invite the reader to see the next chapters for major details about quantitative analysis techniques and efficient solution methods.

5.7 Conclusions

In this chapter we have given an introduction to stochastic process algebra languages describing their development from classical process algebras. In many ways their development has been similar to the development of stochastic Petri net formalisms such as GSPN. In both cases in fact the starting point was an untimed formalism which has been extended to also take into account probabilistic and/or timing information. We have restricted ourselves to the discussion of the main modelling features provided by SPA; (most of) the associated analysis techniques presented later in this book can be applied to SPA too.

In Section 5.6 we informally compared SPA and GSPN and before ending this chapter we briefly summarise some modelling features observing both similarities and important differences.

The model construction facilities of the two formalisms have contrasting strengths and weaknesses. One of the strengths of Petri nets is that causality, conflict and concurrency are clearly depicted within a model and this is true for GSPN models as well. GSPN offer the modeller an explicit notion of state and the graphical notation, which can give an intuitive understanding, and can also be easier to grasp initially. Tool support for GSPN is much more sophisticated largely due to the fact that this formalism has been established for a longer time. This has several consequences: clearly the user is given greater support when using the tool, and the greater support will generally lead to higher confidence on the part of the user.

In contrast, in process algebra based formalisms causality is not exhibited and there is no explicit notion of state. However, the structure within an SPA model is very clearly represented since a compositional structure is apparent in the model. Compositionality provides the possibility of reusability of model components. Modelling is generally an expensive and time-consuming task:

building a new model by composing previously developed components could offer substantial savings. Moreover, the structure makes the model easier to understand, more manageable, and can be exploited for both qualitative and quantitative analysis. Having equivalence relations which are developed and considered to be an integral part of the language means that SPA models can be reasoned about and, in some circumstances, automatically transformed into simpler forms.

If the question is - *what formalism should we use?* - we answer that we do not advocate the adoption of one and the discarding of the other. Instead, we feel that some problem domains will be better suited to expression in one formalism while other problem domains will be better expressed by the other. Thus a modeller would be wise to be familiar with both techniques.

We hope that this introductory presentation gave an intuitive idea of what SPA is, and we invite the reader to read the next chapters bearing in mind that many of the analysis techniques that will be presented in the GSPN context could be (although not always easily) adapted to the SPA domain.

5.8 Bibliographic remarks

Compared to untimed Petri nets, classical process algebras are a more recent formalism since they first appeared in the literature at the end of eighties when the algebraic languages, the Calculus of Communicating Systems (CCS) and the Communicating Sequential Processes (CSP), were introduced. A general introduction to process algebra and CCS may be found in Milner's book [35], probably the best known book on this subject. A good reference for examples and case studies is [14]; a practical introduction to formal methods for specifying concurrent system (based on CCS) may also be found in the recently published book [13]. Concerning CSP, the interested reader may refer to Hoare's book [28].

After the introduction of classical process algebras several extensions, concerning the addition of probabilities and/or timing information, appeared in the literature as we have discussed throughout this chapter.

For timed process algebras, the survey paper by Nicollin and Sifakis [39] provides an overview of the different proposals. More details about the specific languages may be found in the papers [38, 2, 34, 53, 54, 36] already cited in Section 5.4. To the best of our knowledge there is no survey on probabilistic process algebras, and we recommend the papers [32, 45] and [30] for more details about these extensions. Concerning the stochastic extensions, we suggest different references, one for each language we have briefly mentioned in Section 5.5, i.e. [27] for PEPA, [22] for MTIPP, [5] for EMPA, and [7] for MPA.

The development of software tools to support the (S)PA analysis has allowed the usability of the formalism for studying relatively complex systems to be demonstrated. Information about the Concurrency Workbench, a software tool for classical process algebras, may be found in [10]; in the case of SPA we recall the papers [15, 21] which describe more recently developed tools.

The material presented in this chapter covers only some of the basic aspects of the (untimed and timed) algebraic formalism. For a complete overview of its potential, extensions and applications the interested reader can either see the list of papers cited in the bibliography (which is not exhaustive) or the papers published in the proceedings of conferences and workshops on concurrency and related topics in which sections on process algebras are organised. For example, we highlight the International Conference on Concurrency Theory (CONCUR), whose proceedings are published in the series Lecture Notes in Computer Science. Recently, some sections devoted to (stochastic) process algebras have also been organised within the Petri nets conferences (the comparison section is based on the two papers [46, 11] which have been presented at the Petri Nets and Performance Modelling Workshop). As with Petri nets, more abstract work on process algebras also appears in various conferences devoted to theoretical computer science, for example ICALP and LICS.

Starting in 1992, the Process Algebra and Performance Modelling workshop takes place every year with the aim of bringing together researchers and practitioners interested in the development and application of process algebras to performance modelling. The focus is on the interplay between functional and performance analysis in a process algebra setting. Since its inception, the workshop has taken place in Edinburgh, Erlangen, Torino, Twente and Verona. The 1998 workshop, organised by the University of Verona, was a satellite of CONCUR. The 1999 workshop will be held in Zaragoza in conjunction with the International Workshop on Petri Nets and Performance Modelling (PNPM) and the International Conference on Numerical Solution of Markov Chains (NSMC).

Bibliography

[1] J.C.M. Baeten, editor. *Applications of Process Algebra*, volume 17. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

[2] J.C.M. Baeten and J.A. Bergstra. Real time Process Algebra. *Formal Aspects of Computing*, 3, 1981.

[3] C. Baier and H. Hermanns. Weak Bisimulation for Fully Probabilistic Processes. In O. Grumberg, editor, *CAV '97: Computer Aided Verification (Haifa, Israel)*, volume 1254 of *Lecture Notes in Computer Science*, pages 119–130. Springer, 1997.

[4] M. Bernardo, L. Donatiello, and R. Gorrieri. Giving a Net Semantics to Markovian Process Algebras. In *Proc. 6th International Workshop on Petri Nets and Performance Models*, Durham, NC, 1995.

[5] M. Bernardo and R. Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 1998. To appear.

[6] G. Brebner. A CCS-based investigation of deadlock in a multi-process electronic mail system. Technical Report CSR-17-91, The University of Edinburgh, April 1991.

[7] P. Buchholz. Markovian Process Algebra: Composition and Equivalence. In U. Herzog and M. Rettelbach, editors, *Proc.* 2^{nd} *Workshop on Process Algebra and Performance Modelling*, Erlangen, 1994.

[8] L. Chen. *Timed Processes: Models, Axioms and Decidability.* PhD thesis, University of Edinburgh, 1993. CST-101-93.

[9] R. Cleaveland and M. Hennessy. On the consistency of truly concurrent operational semantics. In *Proc. of LICS*. IEEE, 1988.

[10] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15:36–72, January 1993.

[11] S. Donatelli, J. Hillston, and M. Ribaudo. A comparison of Performance Evaluation Process Algebra and Generalized Stochastic Petri Nets. In *Proc. 6th Intern. Workshop on Petri Nets and Performance Models*, Durham, NC, USA, October 1995.

[12] R. Devillers E. Best and J.G. Hall. The Box Calculus: a New Causal Algebra with Multi-level Communication. In *Advances in Petri Nets*, volume 609 of *LNCS*, 1992.

[13] C. Fencott. *Formal Methods for Concurrency.* International Thomson Publishing, 1996.

[14] G.Bruns. *Distributed Systems Analysis with CCS.* Prentice Hall, UK, 1997.

[15] S. Gilmore and J. Hillston. The PEPA workbench: A tool to support a Process Algebra based approach to performance modelling. In *Proc. Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Vienna, 1994.

[16] S. Gilmore, J. Hillston, and L. Recalde. Elementary Structural Analysis for PEPA. Technical Report ECS-LFCS-97-377, Laboratory for Foundations ofComputer Science, University of Edinburgh, December 1997.

[17] S. Gilmore, J. Hillston, and M. Ribaudo. An Efficient Algorithm for Aggregating PEPA Models. Technical report, University of Edinburgh, 1998. Submitted for publication.

[18] U. Goltz. On representing CCS programs by finite Petri nets. In *Mathematical Foundations of Computer Science*, volume 324 of *LNCS*, 1988.

[19] R. Gorrieri, M. Roccetti, and E. Stancampiano. A theory of processes with durational actions. *Theoretical Computer Science*, 1994.

[20] Hans A. Hansson. *Time and Probability in Formal Design of Distributed Systems.* PhD thesis, Dept. of Computer Science, University of Uppsala, September 1991.

[21] H. Hermanns, U. Herzog, U. Klehmet, M.Siegle, and V. Mertsiotakis. Compositional Performance Modelling with the TIPPtool. In *Proc. of 10th Int. Conference on Modelling Techniques and Tool for Computer Performance Evaluation*, LNCS. Springer-Verlag, 1998.

[22] H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic Process Algebras - Between LOTOS and Markov Chains. *Computer Networks and ISDN Systems*, 30 (9-10):901–924, 1998.

[23] H. Hermanns, M. Rettelbach, and T. Weiß. Formal characterisation of immediate actions in SPA with nondeterministic branching. *The Computer Journal*, 38(6):530–541, 1995. Special Issue: Proc. of 3rd Process Algebra and Performance Modelling Workshop.

[24] H. Hermanns and M. Ribaudo. Exploiting Symmetries in Stochastic Process Algebras. In *Proc. of 12th European Simulation Multiconference (Manchester, UK)*. SCS Europe, 1998.

[25] U. Herzog. EXL — Syntax, Semantic and Examples. Technical Report 16/90, Friedrich-Alexander University Erlangen-Nürnberg, IMMD VII, 1990.

[26] J. Hillston. Compositional Markovian modelling using a process algebra. In *Proc. 2nd International Workshop on the Numerical Solution of Markov Chains*, Raleigh, North Carolina, January 1995.

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

[28] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[29] P. Inverardi and M. Nesi. Deciding Observational Congruence of finite state CCS expressions by rewriting. *Theoretical Computer Science*, 139:315–354, 1995.

[30] C-C. Jou and S.A. Smolka. Equivalences, Congruences and Complete Axiomatizations of Probabilistic Processes. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90*, volume 458 of *LNCS*, pages 367–383. Springer-Verlag, August 1990.

[31] Joost-Pieter Katoen. *Quantitative and Qualitative Extensions of Event Structures.* PhD thesis, Centre for Telematics and Information Technology, April 1996.

[32] K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1), 1991.

[33] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationship to Other Models of Concurrency*, volume 255 of *LNCS*. Springer Verlag, 1987.

[34] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25, 1983.

[35] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[36] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *CONCUR '90*, volume 458 of *LNCS*. Springer Verlag, 1990.

[37] I. Rojas Mujica. *Compositional contruction and analysis of Petri nets systems.* PhD thesis, University of Edinburgh, 1998. CST-142-98.

[38] X. Nicollin, J.L. Richier, J. Sifakis, and J. Voiron. ATP: An algebra for timed processes. In *Proc. of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990.

[39] X. Nicollin and J. Sifakis. An overview and synthesis on Timed Process Algebra. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*. Springer Verlag, 1991.

[40] E.-R. Olderog and C.A.R. Hoare. Specification Oriented Semantics for Communicating Processes. *Acta Informatica*, 23:9–66, 1986.

[41] E.R. Olderog. Operational Petri nets semantics for CCSP. In *Advances in Petri Nets*, volume 266 of *LNCS*. Springer Verlag, 1987.

[42] D. Park. Concurrency and Automata on Infinite Sequences. In *Proc.* 5^{th} *GI Conf. on Theoretical Computer Science*, volume 104 of *LNCS*, 1981.

[43] Doron Peled, Vaughan Pratt, and Gerard Holzmann. *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, chapter Debate '90: An electronic discussion on true concurrency. American Mathematical Society, 1997.

[44] G.D. Plotkin. An operational semantics for CSP. Technical Report CSR-114-82, The University of Edinburgh, May 1982.

[45] B. Steffen R. van Glabbeek, S.A. Smolka and C. Tofts. Reactive, generative, and stratified models of probabilistic processes. In *Proc.* 5^{th} *IEEE Int. Symp. on Logic in Computer Science*, 1990.

[46] M. Ribaudo. Stochastic Petri nets semantics for stochastic process algebras. In *Proc.* 6^{th} *Intern. Workshop on Petri Nets and Performance Models*, Durham, NC, USA, October 1995.

[47] S.A. Smolka and B. Steffen. Priority as extremal probability. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90*, volume 458 of *LNCS*. Springer Verlag, 1990.

[48] R. van Glabbeek and F. Vaandrager. Petri nets models for algebraic theory of concurrency. In *PARLE Parallel Architectures and languages Europe*, volume 259 of *LNCS*, 1987.

[49] R.J. van Glabbeek. The linear time – branching time spectrum (extended abstract). In J. C. M. Baeten and J. W. Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR '90, Amsterdam, The Netherlands)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990. Full version available as CWI Amsterdam Report CS-R9029.

[50] R.J. van Glabbeek. The Linear Time – Branching Time Spectrum II: The semantics of sequential systems with silent moves (Extended Abstract). In Eike Best, editor, *Fourth International Conference on Concurrency Theory (CONCUR '93, Hildesheim, Germany)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[51] M. Vardi. Automatic Verification of Probabilistic Concurrent Finite-State Programs. In *Proc. 26th Annual Symposium on Foundations of Computer Science*, pages 327–338, 1985.

[52] G. Winskel. Event structures. In *Petri Nets: Applications and Relationship to Other Models of Concurrency*, volume 255 of *LNCS*. Springer Verlag, 1987.

[53] Wang Yi. Real-Time Behaviour of Asynchronous Agents. In *CONCUR '90*, volume 458 of *LNCS*. Springer Verlag, 1990.

[54] Wang Yi. CCS + Time = an Interleaving Model for Real-Time Systems. In *Proc. of the Eighteenth Colloquium on Automata Languages and Programming*, volume 510 of *LNCS*. Springer Verlag, 1991.