# Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis Using Stochastic Process Algebras*

Norbert Götz    Ulrich Herzog    Michael Rettelbach

Universität Erlangen-Nürnberg
Informatik VII, Martensstr. 3
91058 Erlangen, Germany

**Abstract**

We introduce *Stochastic Process Algebras* as a novel approach for the structured design and analysis of both the functional behaviour and performance characteristics of parallel and distributed systems. This is achieved by integrating performance modelling and analysis into the powerful and well investigated formal description technique of process algebras.

After advocating the use of stochastic process algebras as a modelling technique we recapitulate the foundations of classical process algebras. Then we present extensions of process algebras such that the requirements of performance analysis are taken into account. Examples illustrate the methodological advantages that are gained.

## 1 The Challenge: Constructive Performance Modelling and System Design

### 1.1 Motivation

There are three fundamental categories of attributes indispensable for the viability of any technical system: functionality, performance and economicity [Fer86]. However, it is not unusual for a system to be fully designed and functionally tested before an attempt is made to determine its performance characteristics [Har86]: Redesign of both hardware and software is usually the consequence; this is costly and may cause late system delivery. This is particularly true for real-time systems and a dramatic misdesign of mobile communication ground stations has occurred only recently.

*This research is supported in part by the German National Research Council *Deutsche Forschungsgemeinschaft* under *Sonderforschungsbereich 182* and by the Commission of the European Community as *ESPRIT basic research action QMIPS*, project no. 7269.

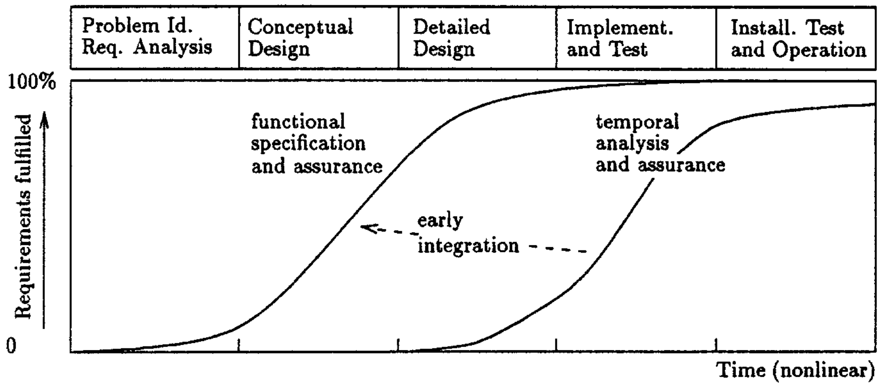


Figure 1: System Life Cycle and Quality Assurance

Figure 1 sketches the phases of any system life cycle and the irresponsible lag between functional and temporal quality specification and assurance. Usually, both functional specification and performance evaluation techniques are separated from each other. System designers use distinct hardware and software specification techniques, while performance assurance is the task of modelling specialists. Such a separation has several advantages, mainly simplicity and understandability. However, to prevent the problems discussed above, performance evaluation has to be intergrated fully into the design process from the very beginning: to eliminate competing approaches, to design and configure components properly, and, last not least, to convince management and customers in due time [Rei91]. Integration of performance analysis into the specification process allows a more accurate description, yields more information about the system behaviour and, again, assures efficient system design.

The need for combined specification methods was already recognized in the seventies. The most successful examples are Stochastic Petri Nets (SPN) [ABC86] and Stochastic Graph Models [ST87, Söt90]. We propose and recommend Stochastic Process Algebras [Her90a, Her90b, Ret91, GHR92, Hil93, SH93] mainly for two important reasons:

- Languages are the principal means of designing hardware and software components: Process algebras are *abstract languages* tailored to the description of parallel and distributed systems; they are well founded, allow a detailed formal specification and support the process of implementation and verification.

- Structuring is the only way to deal with complex systems: Process algebras offer a design methodology, called *constructivity*, that allows to build systematically complex systems from smaller ones. There are operators available for composition as well as mechanisms for abstraction. Finally, process algebras offer an algebraic characterization of equivalent system behaviour.

We emphasize these two important aspects in the next two subsections. Then, in section 2, we give a brief introduction to the basic concepts of process algebras. In section 3 we extend these concepts to Stochastic Process Algebras and present our approach TIPP, a language for *TI*med *P*rocesses for *P*erformance evaluation.

## 1.2 Precise and Modular Description of System Behaviour

Classical queueing and loss models are the best known and most often used performance evaluation techniques. They mainly use a mixture of (almost) standardized box symbols, precisely defined stochastic parameters and colloquially formulated scheduling strategies. These semi-formal techniques have been most successfully applied to describe and analyse the behaviour of many dynamic systems. Modelling, however, the complex interdependencies within distributed and parallel processor systems we need a more precise, expressive and unambiguous description. Formal languages and particularly process algebras (two of them, CSP [Hoa85] and CCS [Mil89], are the basis of the quite remarkable programming languages OCCAM and LOTOS) are best suited and generally accepted as appropriate means.

Stochastic Process Algebras are — as we will elaborate during this tutorial — an extension of these classical abstract languages including random time variables and operators for the description of probabilistic behaviour. Some examples are shown below to illustrate intuitively their expressive power. (Here we assume exponentially distributed time intervals, characterized by their rates; in general these rates just have to be replaced by an appropriate parameter set.)

- The sequential arrival of three different jobs is specified by a process *Jobstream* describing explicitly each arrival point before halting:

$$Jobstream \ := \ (job_1, \lambda_1).(job_2, \lambda_2).(job_3, \lambda_3).Stop$$

- Consequently, a Poisson-arrival process is defined by an infinite sequence of incoming requests, $(in, \lambda).(in, \lambda).(in, \lambda).(in, \lambda).\ \ldots$, which can be formulated recursively:

$$Poisson \ := \ (in, \lambda).Poisson$$

- A service process consisting of an Erlangian distribution of order two is given by:

$$Erl2 \ := \ (end_1, \mu).(end_2, \mu).Stop$$

- Using the probabilistic choice operator, hyperexponentially distributed arrivals are generated by the following process:

$$Hyp2 \ := \ Exp_1 \, [\pi] \, Exp_2 \quad where \quad Exp_i \ := \ (in_i, \lambda_i).Hyp2$$

- Both a precise and concise description of every service or arrival process is possible; this is illustrated by a so-called train-process, which is important for the modelling of file transfers in local area networks. Thereby the overlap and interleaving of different 'trains' is captured by the parallel operator ($\|$):

$$Train \ := \ (lok, \lambda).\{ \ ((wag_1, \mu).(wag_2, \mu). \ \ldots \ (wag_n, \mu).Stop) \ \| \ Train\}$$

Passive actions allow the description of receptive behaviour, i.e. the behaviour of components waiting for activities of some partner, for example:

- The behaviour of a queue, with $i$ jobs already queued, is described by a choice (operator $+$) between waiting for a further job or delivering a job on request to a server:

$$\begin{aligned} Queue_0 &:= (in,-).Queue_1 \\ Queue_i &:= (in,-).Queue_{i+1} + (dlv,-).Queue_{i-1} \qquad i \geq 1 \end{aligned}$$

- An empty server is requesting the delivery of a job, then serves it, and requests again a new job:

$$Server := (dlv,\infty).(srv,\mu).Server$$

The synchronous execution of activities by two components, i.e. joint work or communication, is expressed, again, by the parallel operator, specifying now the synchronous actions explicitly ($\|_{\{...\}}$), for example:

- A queueing system with poisson arrivals and one server is given by

$$QSystem := Poisson \|_{\{in\}} Queue_0 \|_{\{dlv\}} Server$$

  where all component processes are specified as above.

Having seen the expressive power of process algebras, this last example indicates already the second important property, outlined next.

## 1.3 Constructivity: the Basis for Systematic Modelling and System Design

From the very beginning a salient intention of process algebras has been to support a (functional) design methodology which systematically allows to build complex systems from smaller ones. Building blocks for this constructive procedure are *processes* describing the behaviour of the individual system components. Communication activities between these components are accomplished by (synchronous or asynchronous) message passing. There are three important features supporting constructivity:

1. **Composition operators**
   We have already seen several examples for the sequential, parallel or alternative composition of single activities and entire processes. More complex examples will be shown later. The resulting building block is in each case again a process.

2. **Abstraction mechanisms**
   In order to maintain the clearness and transparency of a system design there are operators hiding the internal behaviour of a component. Such a process then shows only its external communication activities to the outside world while internal actions are invisible. Therefore, hiding supports the systematic, clean-cut design and allows the hierarchical structuring of complex systems.

3. **Algebraic characterization of equivalent behaviour**
   Usually there are several possibilities to build a complex system with a specific, intended behaviour. Therefore, alternative designs have to be compared, and components may be interchanged or replaced by others. A major goal of process algebras is the algebraic treatment of system descriptions by means of a simple but powerful calculus. There is a set of transformation rules which allows to build and compare systems with the same behaviour.

Including temporal aspects into such a calculus raises our hope that there is a way to construct and evaluate complex performance models systematically from smaller ones. Nowadays, hierarchical modelling is still an art and only experienced specialists are successful. Stochastic process algebras can offer means for systematic, constructive modelling strategies.

## 2 Introduction to Process Algebras

Classical process algebras — e.g. CSP [Hoa85], CCS [Mil89], ACP [BW90] — have been developed as formal description techniques for systems of processes that run asynchronously in parallel and communicate with each other by synchronously exchanging messages. They showed to be well suited for the description and analysis of so-called 'reactive systems' like robot control systems, operating systems, communication protocols, etc.

The investigation of process algebras has led to comprehensive theories extending classical automata theory in several respects: They provide a *compositional description technique* by defining composition operators for building large and complex descriptions of smaller and simpler ones. Most important are the parallel composition operator for describing synchronized execution of concurrent processes and an operator for abstracting from actions that are considered irrelevant. Different descriptions of the same system can be compared by means of *equivalence relations.* Several notions of equivalence with different granularity have been formally defined and investigated. *Algebraic characterizations* of such equivalence relations yield equational laws that can be used for transforming system descriptions into equivalent ones.

### 2.1 Describing Systems

Modelling the behaviour of a real system means to find a *representation* of the system that covers the relevant aspects of the behaviour and is mechanically tractable. Usually such a representation is given in terms of *states* and *transitions,* where transitions represent state changes that are caused by the execution of system activities. Instead of directly finding such a state-transition based model, process algebras use a clearly defined two-step procedure. First, the system behaviour is described by an abstract language. Second, there is a formally defined semantics, i.e. for each language expression there is a unique interpretation as a state-transition based semantic model. The advantage of such a two-step procedure is obvious: While the first step is design-oriented and user-friendly, the transformation into a state-transition model and its evaluation can be performed automatically.

Within process algebras the basic elements of descriptions are actions and composition operators. Actions describe relevant activities of the system. They are considered to be atomic at the level of abstraction that was chosen for modelling. Descriptions can be composed in order to yield descriptions of more complex systems. Formally, the ways descriptions can be built are defined by a grammar. We assume a fixed set of *action names* $Act := Com \cup \{\tau\}$, where we use $\tau$ as a distinguished symbol for internal, invisible activities and let *Com* be the set of regular, visible activities (the communication actions).

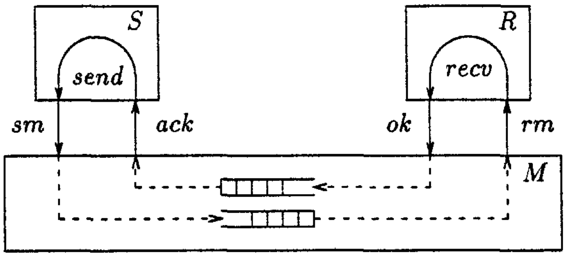
**Definition 2.1** The set $\mathcal{L}$ of terms of the language is given by the grammar

$$\mathcal{P} ::= 0 \mid X \mid a.\mathcal{P} \mid \mathcal{P}+\mathcal{P} \mid \mathcal{P} \|_S \mathcal{P} \mid \mathcal{P}\backslash L \mid recX : \mathcal{P},$$

where $a \in Act$, $S, L \subseteq Com$, and $X \in Var$ with $Var$ being a set of process variables. □

The intuitive meaning of the operators is as follows: 0 denotes a terminated process which cannot perform any actions. The process which performs the action $a$ and then behaves like the process $A$ is denoted by $a.A$. The process $A + B$ can behave either like $A$ or like $B$. The decision is taken as soon as either $A$ or $B$ performs any action. In $A \|_S B$ the component processes $A$ and $B$ proceed concurrently and independently, except for actions in $S$, which must be performed as joint actions. With $A\backslash L$ a set of actions $L$ of the process $A$ is hidden, i.e. these actions become internal and invisible from the outside. Transitions of $A$ due to actions in $L$ will be presented to the environment of $A$ only as anonymous $\tau$-transitions, thus they cannot take part in joint transitions with the environment and can therfore be regarded as invisible. An infinite behaviour is described by a recursive term $recX : A$ with $X$ occurring freely[1] in $A$. An alternative way of writing down a description of recursive behaviour is by using defining equations (cf. the examples of section 1.2). E.g. instead of the term $recX : tick.X$ we can name this process and use this name to indicate the recursion: $Clock := tick.Clock$.

For example consider the task of modelling a send-and-wait protocol. We can describe the protocol by composing three processes: a sender, a receiver, and a communication medium for reliable message transfer. These processes run in parallel but sychronize on actions which describe the handing over of messages from and to the medium.



The sender $S$ repeatedly assembles messages that are to be sent (action *send*). These messages are handed over to the medium (action *sm* - *send message*) and the sender waits for an acknowledge of successful transmission (action *ack*). The receiver $R$ gets a message from the medium (action *rm* - *receive message*), processes it (action *recv*) and returns an acknowledge message (action *ok*).

$$S = recX : send.sm.ack.X \qquad R = recX : rm.recv.ok.X$$

The medium is modelled as two independent one-place-buffers, one for each direction:

$$M = (recX : sm.rm.X) \|_\emptyset (recY : ok.ack.Y)$$

[1]A variable $X$ is called to occur *freely* in a term $A$ if it occurs outside of all subprocesses of the form $recX : B$.

Finally, a desciption of the entire protocol is obtained by a parallel composition of the three processes with appropriate synchronizations:[2]

$$\mathit{SaW\text{-}Prot} \;=\; S \;\|_{\{sm,ack\}}\; M \;\|_{\{rm,ok\}}\; R$$

## 2.2 Operational Semantics

By giving a semantics to our language we define a formal way of creating a behavioural representation of a system from its description. Usually this is done by means of an *operational semantics* that associates a labelled transition system with a process description. Formally, a *transition system labelled over Lab* (the set of labels) is a structure

$$(\, S \,,\, s_0 \,,\, \longrightarrow \,)$$

where $S$ is a set of states, $s_0$ is the initial state and $\longrightarrow \subseteq S \times \mathit{Lab} \times S$ is a transition relation. $(s,l,t) \in \longrightarrow$ means that there is a transition from state $s$ to $t$ with label $l$; this will be written $s \xrightarrow{l} t$.

For presenting a concrete semantics for our language we adopt the Structural Operational Semantics (SOS) style which was introduced by Plotkin [Plo81] and became the most prominent style for process algebra semantics.

With this style, process descriptions and their formal semantics, i.e. their associated labelled transition systems, are formally interweaved by using the set $\mathcal{L}$ of terms of our language as (names of) the states of the transition systems. With a process $P \in \mathcal{L}$ the semantics associates a transition system

$$\mathcal{T}(P) \;=\; (\, \mathcal{L} \,,\, P \,,\, \longrightarrow \,)$$

where $P$ itself represents the initial state and $\longrightarrow \subseteq \mathcal{L} \times \mathit{Act} \times \mathcal{L}$. E.g. for a process that executes the action $a$, afterwards the action $b$, and then stops we will get $\mathcal{T}(a.b.0)$; if this process is alternatively able to execute $c$ and then stop, we will get $\mathcal{T}(a.b.0 + c.0)$.

$$\mathcal{T}(a.b.0) \;=\; \begin{tikzcd} \boxed{a.b.0} \arrow[d, "a"] \\ \boxed{b.0} \arrow[d, "b"] \\ \boxed{0} \end{tikzcd} \qquad \mathcal{T}(a.b.0 + c.0) \;=\; \begin{tikzcd} \boxed{a.b.0 + c.0} \arrow[d, "a"'] \arrow[dd, bend left, "c"] \\ \boxed{b.0} \arrow[d, "b"'] \\ \boxed{0} \end{tikzcd}$$

The main part that remains of the presentation of a semantics is the formal definition of the transition relation. In the SOS-style this is done by means of deduction rules: Every operator of the language constitutes a syntactic form, e.g. $A + B$, of building up descriptions. For every syntactic form we present operational rules defining the transitions that are possible for a process of this form, by referring to the transitions possible for the components of this process. E.g. the rule "*If* $A \xrightarrow{a} A'$

[2]In this case we can omit brackets, but in general brackets are necessary to fix the order of applying the composition operators.

$$\frac{}{a.A \xrightarrow{a} A}\ \langle . \rangle$$

$$\frac{A \xrightarrow{a} A'}{A+B \xrightarrow{a} A'}\ \langle +_l \rangle \qquad\qquad \frac{B \xrightarrow{a} B'}{A+B \xrightarrow{a} B'}\ \langle +_r \rangle$$

$$\frac{A \xrightarrow{a} A'}{A \,\|_S\, B \xrightarrow{a} A' \,\|_S\, B}\ \langle \|_l \rangle \quad (a \notin S) \qquad \frac{B \xrightarrow{a} B'}{A \,\|_S\, B \xrightarrow{a} A \,\|_S\, B'}\ \langle \|_r \rangle \quad (a \notin S)$$

$$\frac{A \xrightarrow{a} A' \qquad B \xrightarrow{a} B'}{A \,\|_S\, B \xrightarrow{a} A' \,\|_S\, B'}\ \langle \| \rangle \quad (a \in S)$$

$$\frac{A \xrightarrow{a} A'}{A\backslash L \xrightarrow{a} A'\backslash L}\ \langle \backslash_{no} \rangle \quad (a \notin L) \qquad \frac{A \xrightarrow{a} A'}{A\backslash L \xrightarrow{\tau} A'\backslash L}\ \langle \backslash_{yes} \rangle \quad (a \in L)$$

$$\frac{A\{recX : A/X\} \xrightarrow{a} A'}{recX : A \xrightarrow{a} A'}\ \langle rec \rangle$$

Figure 2: Operational Semantics Rules

*then* $A + B \xrightarrow{a} A'$" defines that it is possible for a compound process $A + B$ to perform an action $a$ and thereby change into $A'$ if its subprocess $A$ can do so. (A symmetric rule will capture the possible transitions of subprocess $B$.) The general way of writing such rules is

$$\frac{premise_1 \quad \dots \quad premise_n}{conclusion}\ \langle name \rangle \quad (condition)$$

and they are read:

> If *condition* is satisfied, the rule $\langle name \rangle$ can be applied and it can be deduced that *conclusion* holds in case all of the assumptions $premise_1 \dots premise_n$ hold.

Now we can present a semantics for our language:

**Definition 2.2** Let $P \in \mathcal{L}$ be a process description. The semantics of $P$ is defined to be the labelled transition system

$$\mathcal{T}(P) = (\mathcal{L}, P, \longrightarrow)$$

where $\longrightarrow \subseteq \mathcal{L} \times Act \times \mathcal{L}$ is the least relation that satisfies the rules of figure 2. □

The rule for sequential expressions $\langle . \rangle$ is the simplest one. A process of the form $a.A$ can always perform the action $a$ and then changes into $A$. This rule has no precondition and thus serves as an axiom of the deduction system.
A process of the form $A + B$ behaves either like $A$ (according to rule $\langle +_l \rangle$) or like $B$ (according to rule $\langle +_r \rangle$).

If we want to deduce a possible transition of the process $A \parallel_S B$ we can, according to rule $\langle\|_l\rangle$, try to deduce a possible transition for the left subprocess $A$. Let us assume that we can deduce e.g. $A \xrightarrow{a} A'$. In case $a \notin S$, i.e. $a$ does not belong to the synchronization set $S$, the rule $\langle\|_l\rangle$ is applicable and thus we can deduce for $A \parallel_S B$ an $a$-transition to the state $A' \parallel_S B$ where $A$ has changed into $A'$ and $B$ remained unchanged. In case $a \in S$, rule $\langle\|\rangle$ states that it is not sufficient to have just $A \xrightarrow{a} A'$ in order to deduce an $a$-transition for $A \parallel_S B$. There must also be a proof (deduction) that there is an $a$-transition for $B$, say $B \xrightarrow{a} B'$. In this case $A$ and $B$ will move simultaneously to $A' \parallel_S B'$ by executing $a$.
Hiding does not affect transitions due to actions not in $L$ (rule $\langle\backslash_{no}\rangle$), otherwise the label of the transition is changed to the anonymous action $\tau$ denoting an invisible transition (rule $\langle\backslash_{yes}\rangle$). In both cases hiding remains in effect for the successor state.
Finally rule $\langle rec\rangle$ states that a recursive term $recX : A$ behaves exactly like its body $A$, where all the process variables $X$ in $A$ are substituted by the entire recursive term itself.[3]
Note that there is no rule for the terminated process 0, since it does not have any transition that could be deduced.

To see how the operational rules work together and generate the labelled transition system, we deduce a possible transition for the medium of the send-and-wait protocol

$$M \;=\; (recX : sm.rm.X) \parallel_\emptyset (recY : ok.ack.Y) \quad .$$

First we need to find applicable rules by following (top down) the structure of the process description, thereby reducing the term to a sequential form. When more than one rule is applicable we have to choose one of them (and can use the other(s) later for deducing other transitions).

$$\cfrac{\cfrac{sm.rm.(recX:sm.rm.X) \xrightarrow{?} \qquad ?}{recX:sm.rm.X \xrightarrow{?} \qquad ?}\langle rec\rangle}{recX:sm.rm.X \parallel_\emptyset recY:ok.ack.Y \xrightarrow{?} \qquad ?}\langle\|_l\rangle$$

Here we chose to follow the left-hand term of the parallel composition (rule $\langle\|_l\rangle$), unfolded the recursion (rule $\langle rec\rangle$), and finally reached a term in sequential form. Now we apply axiom $\langle . \rangle$, the only rule without further premises, and complete the deduction (bottom up) by matching the single deduction steps with the patterns of the corresponding rules.

$$\cfrac{\cfrac{\cfrac{}{sm.rm.(recX:sm.rm.X) \xrightarrow{sm} rm.(recX:sm.rm.X)}\langle . \rangle}{recX:sm.rm.X \xrightarrow{sm} rm.(recX:sm.rm.X)}\langle rec\rangle}{recX:sm.rm.X \parallel_\emptyset recX:ok.ack.X \xrightarrow{sm} rm.(recX:sm.rm.X) \parallel_\emptyset recY:ok.ack.Y}\langle\|_l\rangle$$

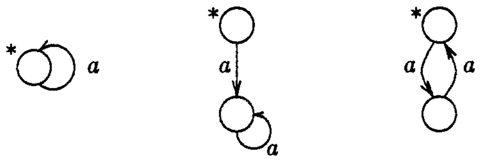
This deduces a possible $sm$-transition for the medium (an $ok$-transition would also be possible and can be deduced starting with rule $\langle\|_r\rangle$ instead of $\langle\|_l\rangle$). The transitions that are possible in the new state, reached by performing $sm$, can be deduced in the same way (there is still the $ok$-transition possible for the right-hand side, and a $rm$-transition for the left-hand side, leading again to the initial state).

[3]The notation $A\{B/X\}$ denotes the *simultaneous* substitution of $B$ for all *free* occurrences of $X$ inside $A$.

This kind of reasoning by means of deduction rules, which are chosen according to the structure of the term, can be automated very easily. Furthermore, every state is represented by a process term containing all the necessary information about the possible transitions leaving this state. This allows to deduce transitions by need, and thus to avoid to create the whole state space in advance when it is not necessary.

## 2.3 Comparing Descriptions

When we model the behaviour of real systems *different representations* can be found, all of them capturing the *'same' behaviour.* E.g. consider the three transition systems (where $*$ indicates the initial state):



The first one executes $a$ infinitely often, the second initially executes an $a$ and then loops forever while executing $a$, and the third toggles between two states by executing $a$. All three transition systems (in fact we can find many more) represent the ability of a system to execute $a$ infinitely often. An external observer watching the three systems and recording the actions they perform would not be able to tell the difference beween these systems from his observations. From the point of view of this observer these systems are considered *equivalent.*

This notion of equivalence of behavioural representations (transitition systems), which is expressed by the metaphor of an observer recording sequences of actions a system performs, can formally be captured as a relation between transition systems. All the possible sequences of actions a system $T = (S, s_0, \longrightarrow)$ can perform, starting from the initial state, are given by the set[4]

$$\begin{aligned} \mathit{traces}(T) \;=\; \{w \in \mathit{Com}^* \mid \exists n \in \mathbb{N}_0.\, \exists l_1, \ldots, l_n \in \mathit{Com}, s_1, \ldots, s_n \in S. \\ w = l_1 \ldots l_n \wedge s_0 \overset{l_1}{\Longrightarrow} s_1 \overset{l_2}{\Longrightarrow} \ldots \overset{l_n}{\Longrightarrow} s_n\} \end{aligned}$$

where $s \overset{a}{\Longrightarrow} t \;:=\; s \xrightarrow{\tau} \ldots \xrightarrow{\tau} s' \xrightarrow{a} t' \xrightarrow{\tau} \ldots \xrightarrow{\tau} t$ is defined to disregard invisible transitions. We can now define

**Definition 2.3** Two labelled transition sytems $T = (S, s_0, \longrightarrow)$ and $T' = (S', s_0', \longrightarrow')$ are *trace equivalent* ($\approx_{\mathrm{tr}}$) iff they have the same set of traces:

$$T \approx_{\mathrm{tr}} T' \;:\Longleftrightarrow\; \mathit{traces}(T) = \mathit{traces}(T')$$

□

Having a means of comparing behavioural representations we can use it to compare *descriptions* of them.

[4]Note that the empty trace is contained in every trace set.

**Definition 2.4** Two process terms $P, P' \in \mathcal{L}$ are *trace equivalent* ($=_{\text{tr}}$) iff their associated transition systems are trace equivalent

$$P =_{\text{tr}} P' \;:\Longleftrightarrow\; \mathcal{T}(P) \approx_{\text{tr}} \mathcal{T}(P')$$

□

E.g. the three transition systems above can be described by the following equivalent terms

$$recX : a.X \quad =_{\text{tr}} \quad a.(recX : a.X) \quad =_{\text{tr}} \quad recX : a.a.X$$

In general we will look for patterns of process terms which are equivalent, thus establishing *equational laws*: There are rather obvious equational laws which reflect certain properties of the language operators, e.g. commutativity and associativity of the operator $+$:

$$\begin{array}{lcl} P + Q & =_{\text{tr}} & Q + P \\ P + (Q + R) & =_{\text{tr}} & (P + Q) + R \end{array}$$

On the other hand there are special laws expressing the inability of an observer to distinguish between certain processes, e.g.

$$\begin{array}{lcl} \tau.P & =_{\text{tr}} & P \\ a.(P + Q) & =_{\text{tr}} & a.P + a.Q \\ a.P \parallel_{\{\}} b.Q & =_{\text{tr}} & a.(P \parallel_{\{\}} b.Q) + b.(a.P \parallel_{\{\}} Q) \end{array}$$

Such a collection of equational laws on processes turns our language into an *algebra*. Once a set of equational laws is presented, two properties of the algebra with respect to the given equivalence notion have to be shown. First, it must be proven that all of the laws are *sound*, i.e. only processes are equated which really have equal trace sets. Second, it must be proven that the set of laws is *complete*, i.e. whenever two processes have the same trace set, they can be shown to be equivalent by purely equational reasoning.

Apart from trace equivalence there is a variety of equivalence notions which have been investigated. In fact, trace equivalence is one of the simplest and coarsest equivalence relations. When we want to compare process descriptions, we must carefully choose an equivalence notion, such that it captures exactly the extent to which we want two processes to be considered equal.

## 2.4 Validation of System Descriptions

A crucial step of modelling real systems is to convince oneself and others that the presented description of a system really captures the system's important properties and represents the relevant aspects of its behaviour properly. To this end several techniques for validating process algebra descriptions have been investigated and are supported by a number of tools. A comprehensive overview of existing tools, their capabilities, and implementation aspects is presented in [IP91].

Validation techniques can be classified into three categories: simulation, equivalence checking, and model checking. They depend strongly on the type of model that is used to represent systems. For this presentation we will only consider validation techniques for systems being represented by transition systems.

**Validation by Simulation** Techniques in this category are based on examining the state space of the system.

Conducting a 'reachability analysis', the state space is searched for states indicating certain unwanted situations. Since with an SOS-style semantics states are represented by process terms, these terms can be checked to support the identification of such situations. As e.g. deadlock states must be distinguished from states representing a regular termination, it is not sufficient just to find states without transitions leaving them; by checking their names, deadlock states like $0 \parallel_{\{a\}} a.0$ can be detected and separated from termination states like $0 \parallel_{\{a\}} 0$.

Furthermore, it is also possible just to list the action sequences the system can perform (certainly almost never exhaustively), or to navigate through the state space, thereby 'testing' the system's ability to react to stimuli from the environment. For performing these kinds of investigations it is not necessary to create the whole state space, but it is sufficient to deduce from the operational semantics rules just the part of the state space needed.

**Validation by Equivalence Checking** Different descriptions of the same system can be proven equivalent. The chosen equivalence notion determines to what extent the behaviours coincide. Equivalence checking is useful in two common situations:

Consider a system consisting of individual subsystems composed by operators of our language, e.g. $A \parallel_S B$. If we want to replace one subsystem, say $B$, by a different, perhaps more efficient, implementation $C$, we expect the new system $A \parallel_S C$ to have the same behaviour as the former one. In case the used equivalence relation ($=_{eq}$) respects the composition operators of the language, i.e. $=_{eq}$ is in fact a congruence relation, we can make use of the following property

$$\forall A, B, C \in \mathcal{L}. \quad B =_{eq} C \implies A \parallel_S B =_{eq} A \parallel_S C.$$

Only the subsystem has to be proven equivalent to the new one replacing it, this is easier than comparing the whole system.

Another situation, where equivalence checking is often used, is the task of verifying that a detailed description of a complex system satisfies an abstract specification. Consider the example of the send-and-wait protocol of section 2.1. If we think of the visible behaviour of the sender and receiver working together being described abstractly by the 'service specification'

$$\mathit{SaW\text{-}Serv} \quad = \quad \mathit{rec}X : \mathit{send}.\mathit{recv}.X$$

then we can prove that this is equivalent to the behaviour of the protocol if we disregard (hide) the internal communication of the sender and receiver with the medium

$$\mathit{SaW\text{-}Serv} \quad =_{tr} \quad (\mathit{SaW\text{-}Prot})\backslash\{\mathit{sm}, \mathit{ack}, \mathit{ok}, \mathit{rm}\}$$

**Validation by Model Checking** A different approach is to use a *logical formalism* for the specification of a system. Relevant properties, which we expect the system to have, are stated one by one as logical expressions and specify conjunctively the system's behaviour. For this purpose temporal logics formalisms can usefully be applied. The task of proving that a behavioural representation (transition system) of a system satisfies its specification is called *model checking* and can be done mechanically.

In contrast to the approach with equivalence checking, where we start with a specification given in form of a process description, which already shows the entire expected behaviour of the final implementation abstractly, the logical approach is more appropriate when such a prototype description is difficult to find and not available from the beginning.

## 3 TIPP — a Language for Timed Processes and Performance Evaluation

### 3.1 Process Algebras and Time

In classical process algebras, only the relative ordering of events is modelled. In order to describe the temporal behaviour of systems, all standard process algebras have been recently extended to *real time process algebras*, which allow to model the *exact timing of events*, and thereby are able to represent time dependent behaviour properly (see [NS91] for an overview). These approaches differ in detail, but have in common the objective to model and analyse the influence of fixed time durations on the functional behaviour rather than to investigate quantitative aspects. Being interested in the analysis of performance characteristics, the concept of *random variables* and *stochastic processes* is still the only feasible way to capture the particulars of a complex system: In many situations there is no exact timing behaviour available because of the complexity of the problem, because of a randomly changing environment, or because of the indeterminacy inherent in any transmission system. Therefore we propose to develop *stochastic process algebras* by incorporating random variables into classical approaches. By doing this we are able to combine the potentials of performance modelling and analysis with the modelling power of standard process algebras.

To the best of our knowledge only two attempts have been made in the past in this direction [NY85, Zic87]. However, the fascinating properties of process algebras are attracting more people now [Hil93, SH93]. In the following sections we report on our research and experiences during the past [Her90b, Ret91, GHR92, GHR93] as well as on ongoing activities.

### 3.2 Stochastic Process Algebras

**The Concept** The basic ideas of our concept are summarized in figure 3. We strictly follow the concept of classical process algebras describing the system behaviour by an abstract language (*system description*) with an underlying state-transition graph representing the exact meaning of the process (*semantic model*). Rather than considering only the functional behaviour we also add temporal information: each activity is described as a pair consisting of its type and time. This additional information in the semantic model allows to evaluate various system aspects:

- *functional behaviour* (e.g. liveness or deadlocks)
- *temporal behaviour* (e.g. throughput or waiting times)

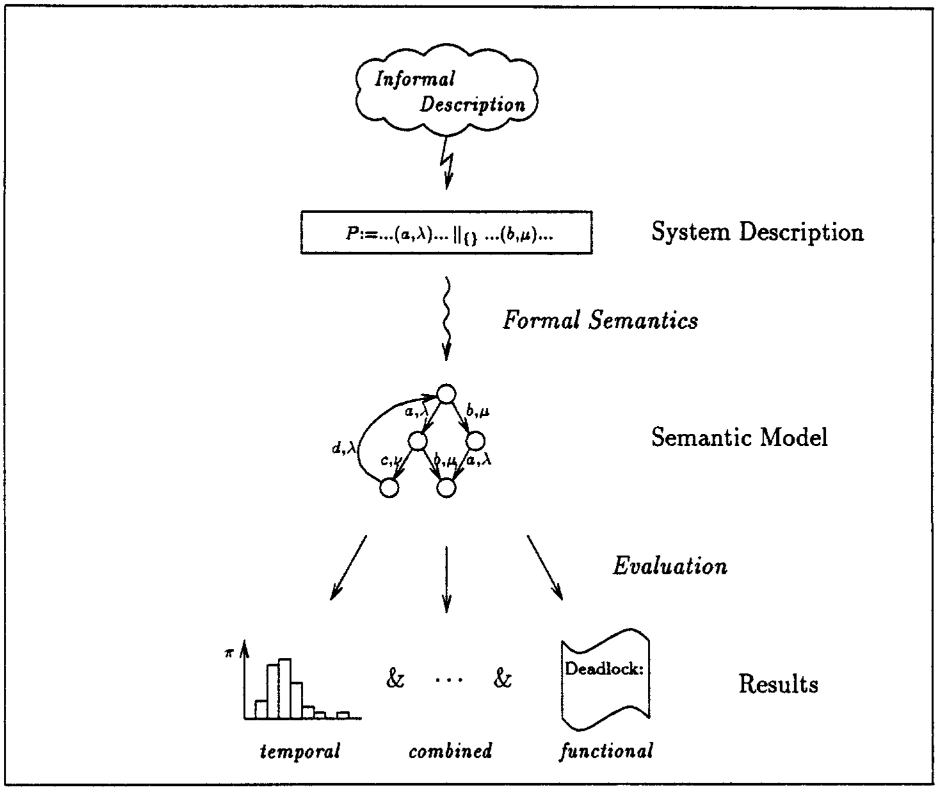


Figure 3: Modelling Procedure

- *combined properties* (such as the duration of certain event sequences, the probability of deadlocks or timeouts, etc.)

Dependent on the stochastic assumptions (e.g. Markovian, semi-Markovian, general) different timing parameters are necessary and the expenditure for evaluations may vary significantly; this is true also for the algebraic characterization of equivalent systems behaviour.

There is not a unique way of solution and there are many possiblities to specify operators, semantic models and equivalences. In principle, however, stochastic process algebras offer the same exciting possiblities as their purely functional forefathers.

**System Description** To incorporate a stochastic time description into process algebras and to guarantee compositional properties, we have chosen the standard concept of stochastic modelling and performance evaluation, a specification by (random) interarrival times: the execution instant of each event is specified by the time interval between the event and its immediate predecessor event; for an illustration cf. figure 4. The events, represented by pairs $(a_i, p_i)$, describe both the functional and temporal behaviour of the process: $a_i$ denotes the type of activities and $p_i$ is a parameter (or parameter set) specifying uniquely the distribution function of the related time intervals:

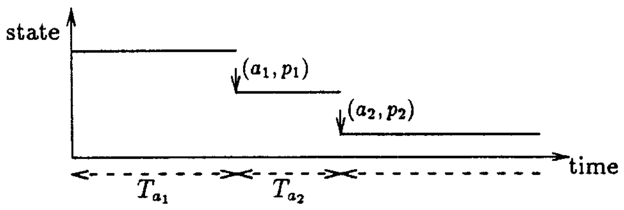


Figure 4: Time Model

- In case of exponentially distributed time intervals the corresponding parameter is the rate $\lambda_i$. (Several examples for such a process description have been shown already in section 1.2.)
- In case of Erlangian distributed intervals two parameters are necessary: the number of phases and the corresponding rate.
- For discrete time distribution functions the random behaviour is given by the appropriate step probabilities.

From a syntactic description we derive a behavioural representation of the system.

**Semantic Model** By means of a formal semantics a system description is translated unambiguously into a transition system representing both the functional and temporal behaviour of the system. For that the semantic must be carefully designed in order to represent properly (either explicitly or implicitly) all of the behavioural information contained in the syntactic description.

- For generally distributed time intervals we preserve this information by so-called 'start-reference'; they allow to solve the problem of residual time intervals in the case of parallel and competing process execution [Ret91, GHR93].
- In case of exponentially distributed time intervals the Markovian property allows a drastic simplification. The structure of the semantic model corresponds to that of the standard Markov chain representation.

Note, however, that the system designer does not see this more or less complicated semantic model; he just formulates the problem to be investigated using the language elements.

In the following two sections we will restrict ourselves to exponentially distributed time intervals and present a stochastic process algebra called Markovian TIPP. Several examples will demonstrate the applicability of our constructive approach to system modelling. Finally, we will show that stochastic process algebras are not limited to the presented version. We briefly sketch how additional language elements like the probabilistic choice operator can be included, and how we can deal with residual execution times when allowing generally distributed time intervalls.

## 3.3 Formal Semantics of Markovian TIPP

In a stochastic process algebra we are dealing with activities which have (random) time duration (described by random variables with a certain time distribution function). Since we will be dealing only with exponential distributions, timed actions can be described by their name and the rate of the exponential distribution function, $(a, \lambda)$. It is often useful to have not only timed actions but also timeless, immediate actions, denoted by $(a, \infty)$. Similar to immediate transitions in Generalized Stochastic Petri Nets [ABC86] they are suitable for the proper modelling of a specific functional behaviour. From a temporal point of view they can be neglected and, by doing so, the number of states of the associated Markov chain model is reduced. Both kinds of actions are called *active*. A third kind of actions, the *passive* actions are introduced to describe subsystems, whose temporal behaviour is not completely determined by their own but by activities of other subsystems communicating with them; e.g. the interarrival time of customers in a queue is determined not by the queue but by the environment (which can be a complex system or be modelled simply by an arrival process) (cf. section 3.4.1). Passive actions, denoted by $(a, -)$, will prove to be very helpful for combining systems by means of synchronized parallel composition.

The formal definition of the language and its operational semantics is completely analogous to the basic process algebra presentation in section 2.2. We assume a fixed set of *action names* $Act := Com \cup \{\tau\}$ with $Com$ being the set of visible actions and $\tau$ a distinguished symbol denoting invisible, internal activities. An action $a$ can either be passive $(a, -)$, immediate $(a, \infty)$, or timed $(a, \lambda)$ with an exponentially distributed duration represented by the parameter $\lambda$.

**Definition 3.1** The set $\mathcal{L}$ of terms of our language is given by the grammar

$$\mathcal{P} ::= 0 \mid X \mid \alpha.\mathcal{P} \mid \mathcal{P} + \mathcal{P} \mid \mathcal{P} \|_S \mathcal{P} \mid \mathcal{P} \backslash L \mid recX : \mathcal{P},$$

where $\alpha \in Act \times Rate$ with $Rate := \{-\} \cup \mathbb{R}^+ \cup \{\infty\}$, $S, L \subseteq Com$, and $X \in Var$ with $Var$ being a set of process variables. □

**Definition 3.2** Let $P \in \mathcal{L}$ be a process description. The semantics of $P$ is defined to be the labelled transition system

$$\mathcal{T}(P) = (\, \mathcal{L} \,,\, P \,,\, \longrightarrow \,)$$

where $\longrightarrow \subseteq \mathcal{L} \times (Act \times Rate \times \{l, r\}^*) \times \mathcal{L}$ is the least relation that satisfies the rules of figure 5. □

The difference between the semantics presented here and that of the basic process algebra (figure 2) is twofold.

First, auxiliary labels ($\in \{l, r\}^*$) are introduced as a technical means to distinguish identical transitions. This is necessary to represent the temporal behaviour correctly. E.g. consider in the basic process algebra the process description $A = b.0 + c.0$. It will yield (according to the rules of fig. 2) two transitions $(A, b, 0), (A, c, 0) \in \longrightarrow$. If we want to change the description to $A' = a.0 + a.0$ where we no longer want to distinguish the activities $b$ and $c$ and call them identically $a$, these two transitions will coincide such that we have only the single transition $(A', a, 0) \in \longrightarrow$. This is all right

$$\frac{}{(a,\lambda).A \xrightarrow{a,\lambda,\varepsilon} A}\ \langle . \rangle$$

$$\frac{A \xrightarrow{a,\lambda,w} A'}{A+B \xrightarrow{a,\lambda,lw} A'}\ \langle +_l \rangle \qquad \frac{B \xrightarrow{a,\lambda,w} B'}{A+B \xrightarrow{a,\lambda,rw} B'}\ \langle +_r \rangle$$

$$\frac{A \xrightarrow{a,\lambda,w} A'}{A \parallel_S B \xrightarrow{a,\lambda,lw} A' \parallel_S B}\ \langle \parallel_l \rangle \quad (a \notin S) \qquad \frac{B \xrightarrow{a,\lambda,w} B'}{A \parallel_S B \xrightarrow{a,\lambda,rw} A \parallel_S B'}\ \langle \parallel_r \rangle \quad (a \notin S)$$

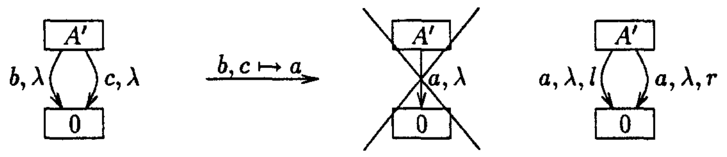
$$\frac{A \xrightarrow{a,\lambda,v} A' \quad B \xrightarrow{a,\mu,w} B'}{A \parallel_S B \xrightarrow{a,max(\lambda,\mu),v\cdot w} A' \parallel_S B'}\ \langle \parallel \rangle \quad (a \in S)$$

$$\frac{A \xrightarrow{a,\lambda,w} A'}{A \backslash L \xrightarrow{a,\lambda,w} A' \backslash L}\ \langle \backslash_{no} \rangle \quad (a \notin L) \qquad \frac{A \xrightarrow{a,\lambda,w} A'}{A \backslash L \xrightarrow{\tau,\lambda,w} A' \backslash L}\ \langle \backslash_{yes} \rangle \quad (a \in L)$$

$$\frac{A\{recX : A/X\} \xrightarrow{a,\lambda,w} A'}{recX : A \xrightarrow{a,\lambda,w} A'}\ \langle rec \rangle$$

Figure 5: Operational Semantics Rules

for the representation of the functional behaviour. For a stochastic process algebra, however, this coincidence in case of $A' = (a,\lambda).0 + (a,\lambda).0$ would lead to a wrong representation of the system's temporal behaviour since the transition rate changes from $2\lambda$ to $\lambda$. Among different possibilities to avoid this coincidence or to adapt the transition rate, we have chosen in TIPP the most tractable solution and use auxiliary labels, $l$ (left) and $r$ (right), to distinguish identical transitions.[5]



Second, and most important from a methodological point of view, the treatment of synchronization fits in the conception of the three kinds of actions we have. If two actions, $(a,x)$ and $(a,y)$, are to be synchronized, it must be considered whether they are active or passive. In order to treat different combinations uniformly, we choose, according to rule $\langle \parallel \rangle$, for the time parameter of the joint action the rate $max(x,y)$, where we extend the function $max$ by defining for passive and immediate actions

$$\forall z \in \{-\} \cup \mathbb{R}^+ \cup \{\infty\}. \quad max(-,z) = z \ \wedge \ max(\infty,z) = \infty$$

[5]In general, in order to distinguish more than two transitions, we concatenate words $w \in \{l,r\}^*$, starting with the empty word $\varepsilon$ in rule $\langle . \rangle$.

This facilitates two important ways of using sychronization while composing descriptions (see also the examples in the next section): either a passive action is synchronized with an active one, $(a, -) \parallel_{\{a\}} (a, \lambda)$ or $(a, -) \parallel_{\{a\}} (a, \infty)$, or two identical actions are synchronized, $(a, x) \parallel_{\{a\}} (a, x)$.[6]

By *synchronizing active and passive actions*, waiting situations can be described. E.g. a processor waiting for a task to execute can be written.

$$Proc := (task, -). \ldots \qquad Workload := \ldots (task, \lambda). \ldots$$
$$System := Proc \parallel_{\{task\}} Workload$$

By *synchronizing identical actions*, system descriptions can be written in an elegant style, the so-called constraint-oriented specification style [V+91]. E.g. consider a workload consisting of three tasks $(t_1, \lambda_1)$, $(t_2, \lambda_2)$ and $(t_3, \lambda_3)$ with the execution constraints '*$t_1$ has to be completed before $t_2$ can start*' and '*$t_2$ has to be completed before $t_3$ can start*'. We can describe this workload by following the structure of its informal specification, combining two constraints on the execution sequence of the three tasks:

$$Workload := (t_1, \lambda_1).(t_2, \lambda_2).0 \parallel_{\{t_2\}} (t_2, \lambda_2).(t_3, \lambda_3).0$$

In this description both subterms (constraints) refer to the *same* task $t_2$. Obviously this description is equivalent to $Workload := (t_1, \lambda_1).(t_2, \lambda_2).(t_3, \lambda_3).0$ but the first one explicitly represents the way it was composed from simpler descriptions. This specification style can usefully be applied not only to timed actions, but also to passive and immediate ones.

## 3.4 Examples

TIPP can be used for the specification and analysis of systems of various kinds. E.g. in [GHR93] we have modelled a *multiple send-and-wait protocol* (cf. section 2.1) using deterministic and exponential time distributions.[7] Here we want to demonstrate how to construct systematically queueing models from basic elements (queues, servers, jobs, etc.) and how the internal structure of the complex jobs can be taken into account.

### 3.4.1 Queueing Systems

We model a simple M/M/n queueing system (see figure 6). A formal description of this system, following the structure of its semi-formal graphical presentation, will be developed from a composition of three parts: an arrival process, a queue, and a number of servers.

The arrival process is modelled as a Poisson stream, creating jobs $a$ (arrivals) with rate $\lambda$.

$$Arr := (a, \lambda).Arr$$

[6]Synchronizing timed actions with the same name but different rates, say $(a, \lambda)$ and $(a, \mu)$, is not explicitly precluded, but we cannot think of any special use for this case; nevertheless the semantics for this case is uniquely defined.

[7]This was carried out with a version of TIPP capable of dealing with general distribution functions (see also section 3.5.2).

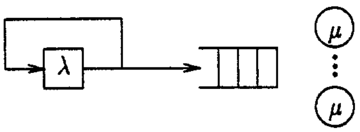


Figure 6: M/M/n queueing system

The queue waits for the arrival of a job, action $(a, -)$, or alternatively delivers a job to a server, if it is requested to do so and it is not empty, action $(d, -)$. In order to get finite models, the queue is bounded to a maximum number of jobs, $maxQ$.

$$\begin{array}{lll} Queue_0 & := & (a,-).Queue_1 \\ Queue_i & := & (a,-).Queue_{i+1} + (d,-).Queue_{i-1} \qquad 0 < i < maxQ \\ Queue_{maxQ} & := & (a,-).Queue_{maxQ} + (d,-).Queue_{maxQ-1} \end{array}$$

The multiserver consists of $n$ independent processing elements. Each processor repeatedly requests a job, being delivered from the queue, action $(d, \infty)$, and processes it with rate $\mu$, action $(p, \mu)$.

$$\begin{array}{lll} Proc & := & (d,\infty).(p,\mu).Proc \\ MultiS & := & \underbrace{Proc \parallel_{\{\}} Proc \parallel_{\{\}} \cdots \parallel_{\{\}} Proc}_{n\text{-times}} \end{array}$$

The complete queueing system is then the synchronized parellel composition

$$M/M/n \;:=\; (Arr \parallel_{\{a\}} Queue_0) \parallel_{\{d\}} MultiS$$

The underlying semantic model of this compact description is determined unambiguously by the semantics of TIPP. For $n = 2$ the model is depicted in figure 7, where the horizontal position of the nodes indicates the number of tasks in the system (0 - $maxQ$+2) and the vertical position represents the number of processors busy (0 – $n$). This model can be transformed quite easily into a Markov chain by eliminating

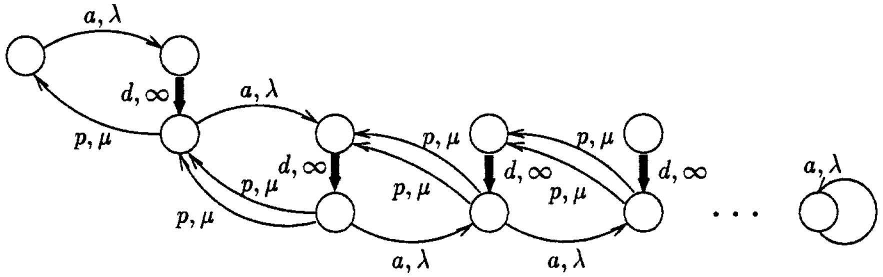


Figure 7: Semantic model of the system $M/M/n$

immediate transitions, dropping the action names, and joining multiple transitions by adding their rates.

### 3.4.2 Task Graphs

Jobs consisting of a number of interdependent tasks can be represented graphically by task graphs. E.g. consider a *Job* with tasks $t_1, \ldots, t_6$ and a precedence relation depicted in figure 8. The task graph can be assumed to be developed from the

$$\begin{tikzcd}
 & t_2 \arrow[r] \arrow[drr] & t_4 \arrow[dr] & \\
t_1 \arrow[ur] \arrow[dr] & & & t_6 \\
 & t_3 \arrow[r] & t_5 \arrow[ur] &
\end{tikzcd}$$

Figure 8: Task graph

combination of several precedence constraints, e.g.

$$\begin{aligned}
C_1 &:= (t_1, \lambda_1).(t_2, \lambda_2).(t_4, \lambda_4).(t_6, \lambda_6).0 \\
C_2 &:= (t_1, \lambda_1).(t_3, \lambda_3).(t_5, \lambda_5).(t_6, \lambda_6).0 \\
C_3 &:= (t_2, \lambda_2).(t_5, \lambda_5).0 \\
\mathit{Job} &:= (C_1 \parallel_{\{t_1, t_6\}} C_2) \parallel_{\{t_2, t_5\}} C_3
\end{aligned}$$

### 3.4.3 Mapping Workload onto a Multiserver

Now we want to present a more advanced example combining the two previous examples and describe the treatment of complex jobs with several tasks in a queueing network.

First we need to consider the effects of synchronization in some more detail. In general synchronization of system activities, modelled by timed actions, and system resources, modelled by passive actions, requires some care. Consider e.g. a system with a workload consisting of two independent tasks and a single processor, which, for the sake of this example, serves just one of the tasks and then stops.

$$\mathit{Workload} := (\mathit{task}, \lambda_1).0 \parallel_{\{\}} (\mathit{task}, \lambda_2).0 \qquad \mathit{Proc} := (\mathit{task}, -).0$$
$$\mathit{System} := \mathit{Workload} \parallel_{\{\mathit{task}\}} \mathit{Proc}$$

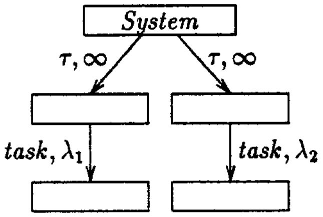
Unfortunately, describing the system in this way is incorrect, since the assigned behavioural representation contradicts our intuition about the system behaviour.

$$\begin{tikzcd}
 & \boxed{\mathit{System}} \arrow[dl, "{\mathit{task}, \lambda_1}"'] \arrow[dr, "{\mathit{task}, \lambda_2}"] & \\
\boxed{(0 \parallel_{\{\}} (t, \lambda_2)) \parallel_{\{t\}} 0} & & \boxed{((t, \lambda_1) \parallel_{\{\}} 0) \parallel_{\{t\}} 0}
\end{tikzcd}$$

Here the overall transition rate from the initial state to a terminal state is $\lambda_1 + \lambda_2$. The reason is, that the *potential* parallelism of the workload is represented as *actual* parallelism rather than being eliminated.

A correct description would be to eliminate the parallelism of the workload by making first a decision, which of the tasks is to be started.

$$\begin{array}{lcl} \mathit{Workload} & := & (\mathit{start\_t}, -).(\mathit{task}, \lambda_1).0 \parallel_{\{\}} (\mathit{start\_t}, -).(\mathit{task}, \lambda_2).0 \\ \mathit{Proc} & := & (\mathit{start\_t}, \infty).(\mathit{task}, -).0 \\ \mathit{System} & := & (\mathit{Workload} \parallel_{\{\mathit{start\_t},\ \mathit{task}\}} \mathit{Proc})\backslash\{\mathit{start\_t}\} \end{array}$$



This leads to a branching structure in the transition system with two immediate transitions, representing a choice between the two alternative tasks with equal probability.

Now we can adapt the description of the task graph of the previous section for being included in the queueing network example (additionally, an immediate action *job_end* indicates the termination of the job).[8]

$$\begin{array}{lcl} C_1 & := & (\mathit{start\_t}_1, -).(\mathit{task}_1, \lambda_1).\ (\mathit{start\_t}_2, -).(\mathit{task}_2, \lambda_2). \\ & & \quad (\mathit{start\_t}_4, -).(\mathit{task}_4, \lambda_4).\ (\mathit{start\_t}_6, -).(\mathit{task}_6, \lambda_6). \\ & & \qquad (\mathit{job\_end}, \infty).\ 0 \\ C_2 & := & (\mathit{start\_t}_1, -).(\mathit{task}_1, \lambda_1).\ (\mathit{start\_t}_3, -).(\mathit{task}_3, \lambda_3). \\ & & \quad (\mathit{start\_t}_5, -).(\mathit{task}_5, \lambda_5).\ (\mathit{start\_t}_6, -).(\mathit{task}_6, \lambda_6).\ 0 \\ C_3 & := & (\mathit{task}_2, \lambda_2).\ (\mathit{start\_t}_5, -).\ 0 \\ \mathit{Job} & := & (C_1 \parallel_{\{\mathit{start\_t}_1, \mathit{task}_1, \mathit{start\_t}_6, \mathit{task}_6\}} C_2) \parallel_{\{\mathit{task}_2, \mathit{start\_t}_5\}} C_3 \end{array}$$

The queueing network is composed of the same parts as before: arrival process, queue and server. The arriving jobs are all of the same kind. As jobs consist of individual tasks, the processors of the server must be able to process any of these different tasks.[9]

$$\begin{array}{lcl} \mathit{Proc} & := & \sum_{i=1}^{6} (\mathit{start\_t}_i, \infty).(\mathit{task}_i, -).\mathit{Proc} \\ \mathit{MultiS} & := & \mathit{Proc} \parallel_{\{\}} \mathit{Proc} \parallel_{\{\}} \cdots \parallel_{\{\}} \mathit{Proc} \end{array}$$

The complex structure of a job delivered from the queue makes it necessary to supplement the multiserver with a *scheduler* which mediates between the queue and the processors.

$$\mathit{Scheduler} := (\ (d, \infty).\mathit{Job} \parallel_{\{\mathit{job\_end}\}} (\mathit{job\_end}, -).\mathit{Scheduler}\ )\backslash\{\mathit{job\_end}\}$$

After requesting a job from the queue the scheduler activates it, so that the individual tasks of the job can be executed on the processors according to the precedence relation

[8]The description given in the previous section is all right if we want to analyse the task graph for its own, exploiting the full inherent parallelism.

[9]We will write $\sum_{i=1}^{n} P_i$ abbreviating the $n$-fold alternative composition $P_1 + \cdots + P_n$.

specified in the description of *Job*. At the end of the job an internal (hidden) signal (*job_end*) (see description of $C_1$ above) reactivates the scheduler.

The complete system is then described by

$$(Arr \parallel_{\{a\}} Queue_0) \parallel_{\{d\}} (Scheduler \parallel_S MultiS)$$

with $S = \{start\_t_i, task_i \mid i = 1, \ldots, 6\}$.

This description can easily be modified such that certain tasks are only executed on certain processors. E.g. a server with two processors $Proc_1$, $Proc_2$, where $Proc_1$ executes only $task_1$ and $task_2$ and $Proc_2$ the others, is described by

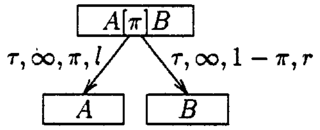
$$\begin{array}{lcll} Proc_1 & := & \sum_{i \in T_1}(start\_t_i, \infty).(task_i, -).Proc_1 & T_1 = \{1,2\} \\ Proc_2 & := & \sum_{i \in T_2}(start\_t_i, \infty).(task_i, -).Proc_2 & T_2 = \{3,4,5,6\} \\ MultiS & := & Proc_1 \parallel_{\{\}} Proc_2 & \end{array}$$

## 3.5 Extensions

### 3.5.1 Probabilistic Choice

For describing alternative behaviour with fixed branching probabilities we need to extend our language by a new operator, the probabilistic choice operator $A[\pi]B$. We will present here a rather staightforward way of incorporating it into the semantics.

A probabilistic choice $A[\pi]B$ is considered to be resolved instantaneously and completely internal. The decision is taken in favour of $A$ with probability $\pi$ and in favour of $B$ with $1 - \pi$. This is represented in a transition system by branching off two immediate transitions which are labelled additionally with the corresponding probability.

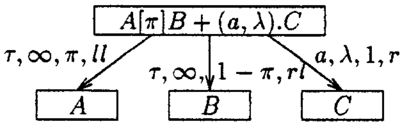


Semantically, this requires to introduce new rules and to extend the labels of transitions. Labels are now quadrupels $Act \times Rate \times \mathbb{R}_{[0,1]} \times \{l, r\}^*$ with $\mathbb{R}_{[0,1]}$ being the closed interval of real numbers from 0 to 1. There are two new rules:

$$\frac{}{A[\pi]B \xrightarrow{\tau,\infty,\pi,l} A}\ \langle \pi_l \rangle \qquad \frac{}{A[\pi]B \xrightarrow{\tau,\infty,1-\pi,r} B}\ \langle \pi_r \rangle$$

These rules have no preconditions, since the behaviour of a probabilistic choice does not depend on the behaviour of its constituent parts.

This treatment of probabilistic choice, however, interferes with the intuition of the competitive choice. E.g. in the system $A[\pi]B + (a, \lambda).C$

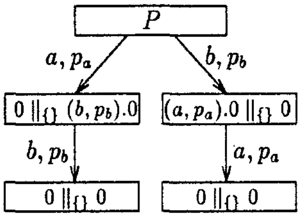


the $a$-transition competes with two immediate transitions and will therefore never be executed. Such situations can be ruled out in advance by syntactically demanding that a probabilistic choice must be *'guarded'*, i.e. a subterm $A[\pi]B$ can only occur inside a term if it is sequentially preceeded by an action. This can be expressed formally by distinguishing syntactic categories in the grammar of the syntax definition.

$$\begin{array}{lcl} \mathcal{P} & ::= & \mathcal{Q} \mid \mathcal{P}[\pi]\mathcal{P} \mid recX : \mathcal{P} \\ \mathcal{Q} & ::= & 0 \mid X \mid \alpha.\mathcal{P} \mid \mathcal{Q} + \mathcal{Q} \mid \mathcal{Q} \parallel_S \mathcal{Q} \mid \mathcal{Q}\backslash L \end{array}$$

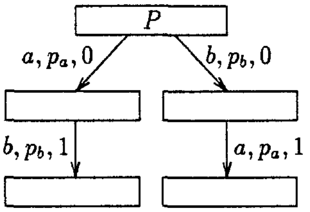
### 3.5.2 General Distribution Functions

Dealing with time distribution functions other than exponential ones requires a careful semantic treatment of residual execution times. Consider a process $P = (a, p_a).0 \parallel_{\{\}} (b, p_b).0$, where $p_a$, $p_b$ are suitable parameters describing the time distributions of the associated random variables $T_a$, $T_b$ (cf. figure 4). With the semantics of Markovian TIPP we would get the following transition system:



This representation would be incorrect, since the time distribution for the transition $0 \parallel_{\{\}} (b, p_b).0 \xrightarrow{b, p_b} 0 \parallel_{\{\}} 0$ is different from the one denoted by the parameter $p_b$: Action $b$ was already enabled in state $P$, but action $a$, an action of a parallel and totally independent process, happened to be executed first. Thus the correct time distribution is $P(T_b \leq t + T_a \mid T_b > T_a)$, with $T_a$, $T_b$ being random variables recording the (total) execution time of $a$ and $b$, respectively.

In order to represent residual execution times properly in the semantic model, there are two principal possiblities, it can be done either explicitly or implicitly. An explicit representation would adjust the time parameter of the transition, such that it denotes the correct residual time distribution. This treatment has several disadvantages; therefore we have chosen an implicit representation [GHR93]. We attach to every transition a so-called *start reference* that points to that state in the transition system where the corresponding action was enabled. Start references are represented by natural numbers, counting the number of actions that have happened since the considered action was ready to be performed. Thus it indicates the actual starting point of the time interval associated with the action.



When it comes to analysing the performance of a system, the start references provide sufficient information to determine the actual time distribution. Thus it is avoided to determine residual time distributions unnecessarily.

## 4 Conclusions and Prospects

Usually both functional specification and performance modelling are separated from each other. Simplicity and understandability are strong arguments. However, in many situations an integrated approach is necessary and advantageous: to capture the particulars of parallel and distributed systems, to support hierarchical modelling, and to improve design productivity.

We introduced Stochastic Process Algebras as a novel approach for structured design and analysis, taking into consideration both the fuctional and temporal behaviour: The designer specifies the components of a system and their interactions by means of the language TIPP. Then this syntactic representation is (automatically) transformed into a semantic model which may be analysed in various directions: (1) functional characteristics, (2) performance characteristics, and (3) quality aspects influenced by both functional and temporal behaviour.

Combining performance modelling and analysis with the well investigated formal description technique of process algebras provides a rich source of further developments. A lot can be gained by exploiting the comprehensive theories of process algebras, which have been developed mainly during the last decade, and make their results and techniques available for performance modelling. Just to indicate a few ideas we are thinking of: The mapping of a workload onto a multiprocessor (cf. section 3.4.3) could be described more generally by parametrizing the parallel composition by a synchronization function [BW90]. Comparison of process descriptions can be done not only by equivalence relations; an order relation [Hen88] comparing e.g. the performance of processes would be very much desirable. Abstraction mechanisms, like the hiding operator, must be extended to include also the temporal behaviour; this could yield an approach to treat the problem of hierarchical modelling.

Our work is going on in three main directions: We are investigating the theoretical foundations ([Do93],[Her93]), we are searching for efficient evaluation algorithms, and, last not least, modelling the behaviour of real systems has a strong influence on the first two directions.

## References

[ABC86] M. Ajmone Marsan, G. Balbo, and G. Conte. *Performance Models of Multiprocessor Systems.* MIT Press, 1986.

[BW90] Jos Baeten and Peter Weijland. *Process Algebra.* Cambridge University Press, 1990.

[Do93] Hu Trung Do. *Entwurf von Prozeßsprachen zur Leistungsbewertung.* Diplomarbeit, Universität Erlangen, 1993.

[Fer86] D. Ferrari. Considerations on the Insularity of Performance Evaluation. *IEEE Transactions on Software Engineering*, SE–12(6):678–683, June 1986.

[GHR92] Norbert Götz, Ulrich Herzog, and Michael Rettelbach. TIPP — a language for timed processes and performance evaluation. Interner Bericht IMMD7-4/92, Universität Erlangen, March 1992.

[GHR93] Norbert Götz, Ulrich Herzog, and Michael Rettelbach. TIPP — introduction and application to protocol performance analysis. In *Formale Beschreibungstechniken für verteilte Systeme*, Munich, to appear 1993. FOKUS series, Saur publishers.

[Har86] C. Harvey. Performance engineering as an integral part of system design. *British Telecom Technology Journal*, 4(3):143–147, July 1986.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Her90a] Ulrich Herzog. EXL: Syntax, semantics and examples. Interner Bericht IMMD7-16/90, Universität Erlangen, November 1990.

[Her90b] Ulrich Herzog. Formal description, time and performance analysis — a framework. In *Entwurf und Betrieb verteilter Systeme.* Springer, 1990. Informatik Fachberichte 264.

[Her93] Holger Hermanns. *Semantik für Prozeßsprachen zur Leistungsbewertung.* Diplomarbeit, Universität Erlangen, to appear 1993.

[Hil93] J. Hillston. PEPA: Performance Enhanced Process Algebra. Technical Report CSR-24-93, University of Edinburgh, March 1993.

[Hoa85] Charles Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[IP91] Paola Inverardi and Corrado Priami. Evaluation of tools for the analysis of communicating systems. *EATCS Bulletin*, 45:158–185, 1991.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Real-Time: Theory in Practice*, pages 526–548. Springer LNCS 600, 1991.

[NY85] N. Nounou and Y. Yemini. Algebraic specification-based performance analysis of communication protocols. In *Protocol Specification, Testing and Verification*, pages 541–560. Elsevier Publishers, 1985.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

[Rei91] M. Reiser. A quarter century of performance evaluation — impact on science and engineering. In *Proceedings of the IEEE CompEuro '91*, pages 885–887. IEEE, May 1991.

[Ret91] Michael Rettelbach. *Leistungsbewertung mit Prozeßalgebren.* Diplomarbeit, Universität Erlangen, 1991.

[SH93] Ben Strulo and Peter Harrison. Process algebra for discrete event simulation. Technical report, Imperial College, March 1993.

[Sõt90] F. Sõtz. A method for performance prediction of parallel programs. In Burkhart, editor, *CONPAR 90-VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 98–107. Springer LNCS 457, 1990.

[ST87] R. A. Sahner and K. S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Transactions on Software Engeneering*, SE-13(10):1105–1114, 1987.

[V+91] Chris Vissers et al. Specification styles in distributed system design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

[Zic87] J. J. Zic. Extensions to communicating sequential processes to allow protocol performance specification. *ACM Computer Communication Review, Special Issue: SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology*, 17(5):217–227, 1987.