Technique and tool for symbolic representation and manipulation of stochastic transition systems[*]

Markus Siegle
Universität Erlangen-Nürnberg, IMMD VII
Martensstraße 3, 91058 Erlangen
siegle@informatik.uni-erlangen.de

May 22, 1998

Abstract: This paper presents a new approach to the compact symbolic representation of stochastic transition systems. We introduce Decision Node BDDs, a novel stochastic extension of BDDs which preserves the strucure and properties of purely functional BDDs. It is shown how parallel composition of stochastic transition systems can be performed on the basis of this new data structure. Furthermore, we discuss state space reduction by Markovian bisimulation, also based on symbolic techniques.

1 Introduction

In many areas of system design and analysis, there is the problem of generating, manipulating and analysing large labelled transition systems (LTS), i.e. transition systems with a large number of states and a large number of transitions. Such transition systems are often very difficult (or even impossible) to handle in practice, due to memory limitations.

In this paper, the focus is on *stochastic* LTSs, where each transition is associated with a stochastic delay. Such stochastic LTSs (SLTS) occur during performance evaluation and performability analysis of distributed systems. For example, stochastic LTSs are generated during the analysis of stochastic process algebra (SPA) models, by applying semantic rules to SPA specifications. Under certain conditions, disregarding part of their information contents, SLTSs can be interpreted as Markov chains, rendering them amenable to numeric analysis methods.

We propose a novel approach to SLTS representation and manipulation which is based

[*]Universität Erlangen-Nürnberg, TR IMMD 7, 2/98, March 1998

on symbolic techniques. Our work has been motivated by the fact that, in recent years, the problem of large LTS analysis has been very successfully approached by using symbolic representations, in particular binary decision diagrams (BDD). Most of this work took place in the context of formal verification and model checking, i.e. it deals exclusively with functional behaviour, see e.g. [5, 8, 10]. This experience showed that symbolic representations make it possible to handle larger state spaces than traditional methods.

The success of symbolic techniques for functional analysis induced us to experiment with BDD-based representations of *stochastic* LTS. Representation of non-functional information in a symbolic form is possible [6], but representation of stochastic LTS has not got much consideration in the past. Among the few publications in this line are [14] and [9].

The contribution of this paper is as follows: A novel data structure, DNBDD, is introduced, which can capture not only functional, but also *temporal (stochastic)* information. This data structure is tailored for SLTS, it allows a very compact representation. It is shown that known algorithms for BDDs can be adapted and enhanced for the new data structure. Furthermore, we describe a minimisation algorithm for stochastic LTS which is based on the concept of Markovian bisimulation and works entirely on the new data structure. The paper shows the feasability and the advantages of the new method. We also briefly discuss the practical implementation of our concepts in the form of a prototype tool.

In a previous publication we had informally described the basic idea and given an intuitive overview of DNBDDs [27].

2 Process algebras and transition systems

2.1 Stochastic process algebras

Process algebras are languages for specifying the behaviour of concurrent processes [20, 23]. In recent years, process algebras have enjoyed increasing popularity. In particular, they have been extended in order to describe not only functional, but also temporal properities of processes. For the purpose of performance and dependability analysis, stochastic concepts have been incorporated, leading to stochastic process algebras (SPA).

We briefly introduce a simple SPA, which is defined by the following grammar:

$$P \ ::= \ 0 \quad | \quad X \quad | \quad (a,\lambda).P \quad | \quad P+P \quad | \quad P \parallel_S P \quad | \quad recX : P \quad | \quad P \setminus L$$

The non-terminal symbol P represents a process. The symbol $a \in Act$ is an action with its associated rate λ. The rate is interpreted as the parameter of an exponential distribution. Thus, the stochastic delay after which an action takes place is specified through a rate parameter. 0 denotes a stopped process. $X \in Var$ is a process variable. The operators for prefixing, choice, parallel composition, recursion and hiding have the usual meaning (cf., e.g. [16, 13]). $S \subseteq Act$ is the set of synchronising actions for parallel composition of two processes. $L \subseteq Act$ is the set of actions which are hidden from the environment.

During the analysis of a process algebra description, the application of semantic rules leads to labelled transition systems. In case of stochastic process algebras, *stochastic* labelled transition systems are generated. When modelling a real-life system, it is often the case that the transition systems generated from the SPA description become very large. They may become even intractable due to memory and CPU limitations. This well-known state space explosion problem has been addressed in the past by many researchers (some of our own work on state space reduction can be found in [25, 26]). In the present paper we investigate a new symbolic approach to the storing and manipulating of large stochastic transition systems.

2.2 Stochastic Transition Systems

Informally, a transition system consists of states and transitions between states. The transitions are labelled with symbols from a set L which usually correspond to the set of actions *Act*. In case of *stochastic* transition systems, each transition has as a second attribute a real number, the *rate* of the transition. A SLTS can be graphically interpreted as a directed graph (potentially with a distinguished initial node) whose edges are labelled with tuples from $L \times \mathbb{R}$. Fig. 1 shows an example stochastic LTS.

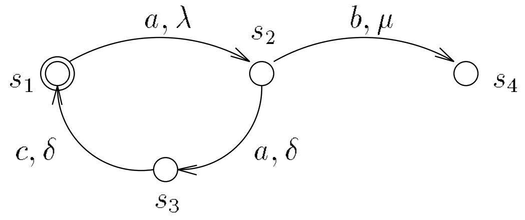


Figure 1: example of a stochastic labelled transition system (SLTS)

We now give a formal definition of STLS:

Def: *Stochastic Labelled Transition System (SLTS)*
Let $S = \{s_1, s_2, \ldots\}$ be a finite set of states, s_1 being the initial state.
Let $L = \{l_1, l_2, \ldots\}$ be a set of labels.
Let f be a function

$$f : S \times L \times S \rightarrow \mathbb{R}$$

We call $\mathcal{T} = (S, L, f)$ a *stochastic Labelled Transition System.*
If $f(x, a, y) = \lambda \neq 0$ we say that there is an a-transition from state x to state y with rate λ. We write $x \overset{a,\lambda}{\rightarrow} y$. If $f(x, a, y) = 0$ we say that there is no a-transition from x to y. ■

Remark: Instead of the above function $f : S \times L \times S \rightarrow \mathbb{R}$, most definitions of SLTS use a transition relation $\rightarrow \subseteq S \times L \times \mathbb{R} \times S$, which allows multiple transitions with the same action label between a fixed pair of states. We chose the above definition in order to explicitely exclude such multiple transitions. Instead of two separate tansitions $x \overset{a,\lambda}{\rightarrow} y$ and $x \overset{a,\mu}{\rightarrow} y$, we would represent this situation by $x \overset{a,\lambda+\mu}{\rightarrow} y$.

Note that in our definition the set of states S is assumed to be finite. Finiteness of the state space is a prerequisite for the symbolic encoding of states and transitions which is described in the next section.

The real-valued rates specify the time spent in a particular state, which is a random value, drawn from an exponential distribution. The mean of this distribution is given by the inverse of the sum of all rates of transitions leaving that state. For example, in fig. 1, the mean time spent in state s_1 is $1/\lambda$, and the mean time spent in state s_2 is $1/(\mu + \delta)$. There is a close relation between SLTSs and continuous time Markov chains (CTMC). The CTMC corresponding to a SLTS is obtained by abstracting from the action labels. Vice versa, a SLTS can be considered a CTMC whose transitions carry additional labels taken from a set of actions.

3 Symbolic representation of transition systems

3.1 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [4, 1] is a symbolic representation of a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$. Its graphical interpretation is a rooted directed acyclic graph (DAG) with one or two (terminal) leaf nodes. The graph is essentially a collapsed binary decision tree in which isomorphic subtrees are merged and don't care nodes are skipped. As a simple example, fig. 2 (left) shows the BDD for the function $\overline{a}\ t + a\ s\ \overline{t}$. The function value for a given truth assignment can be determined by following the corresponding edges (one-edges drawn solid, zero-edges dashed) from the root until a terminal node is reached. In the graphical representation of a BDD, for reasons of simplicity, the terminal false node and its adjacent edges are usually omitted, see fig. 2 (right).

Next, we give a formal definition of BDD:

Def: *Binary Decision Diagram*
A (reduced, ordered) *Binary Decision Diagram* is defined by

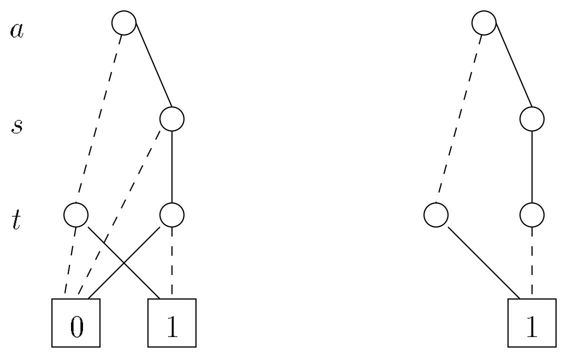


Figure 2: BDD for $\overline{a}\, t + a\, s\, \overline{t}$, simplified graphical representation (right)

- a set of nodes $Nodes = \{n_1, \ldots, n_k\} = T \cup NT$, where T (NT) is the set of (Non-) terminal nodes, $|Nodes| \geq 1$, $T \subseteq \{false, true\}$ (instead of $\{false, true\}$ we often use $\{0, 1\}$),
- a set of Boolean variables $Vars = \{v_1, \ldots, v_n\}$ with a fixed ordering relation "<", such that $v_1 < \ldots < v_n$,
- a function $var : NT \rightarrow Vars$,
- a function $low : NT \rightarrow Nodes$,
- a function $high : NT \rightarrow Nodes$,

with the following constraints:

1. $\forall x \in NT : low(x) \in T \vee var(low(x)) > var(x)$
   $\forall x \in NT : high(x) \in T \vee var(high(x)) > var(x)$ (respect ordering relation),
2. $\forall x \in NT : low(x) \neq high(x)$ (no redundant (don't care) nodes),
3. $\forall x, y \in NT : \quad var(x) \neq var(y) \vee low(x) \neq low(y) \vee high(x) \neq high(y)$ (no isomorphic nodes),
4. $1 \leq |T| \leq 2$ (one or two terminal nodes),
5. $\nexists\, x_1, x_2 \in Nodes : \quad low^{-1}(x_1) = high^{-1}(x_1) = \emptyset \wedge low^{-1}(x_2) = high^{-1}(x_2) = \emptyset$
   (only one node without predecessor, i.e. only one root node). ■

A BDD unambiguously defines a Boolean function. The definition is based on the so-called Shannon expansion which states that

$$f(v_1, \ldots, v_n) = v_1 \cdot f(1, v_2, \ldots, v_n) + \overline{v_1} \cdot f(0, v_2, \ldots, v_n).$$

Def: *Boolean function $Bool(x)$*
The *Boolean function $Bool(x)$ represented by a BDD-node $x \in Nodes$* is recursively defined as follows:

- if $x \in T$ then $Bool(x) = x$, i.e. either *true* or *false*,
- else (if $x \in NT$) $Bool(x) = \overline{var(x)} \cdot Bool(low(x)) + var(x) \cdot Bool(high(x))$. ■

For convenience, we define the following function:

Def: *Boolean result value $Bool\_res(b_m, \ldots, b_n)$*
Let $b_m, \ldots, b_n$ $(1 \leq m \leq n, b_i \in \{true, false\})$ be a fixed assignment to the Boolean variables $v_m, \ldots, v_n$. Let $x \in NT$ and $var(x) = v_m$. The Boolean result value of the function represented by node x under this assignment is
$Bool\_res(b_m, \ldots, b_n) = Bool(x)|_{v_m=b_m,\ldots,v_n=b_n}$. ■

Most times one is interested in the case $m = 1$, i.e. the case in which x corresponds to the BDD root.

It is known, that BDDs provide a canonical representation for Boolean functions, i.e. a given Boolean function has a unique BDD representation (assuming a fixed ordering of the Boolean variables) [4]. For this reason, some computationally hard problems (e.g. satisfyability, test-for-tautology, equivalence) can be solved in constant or linear time, once the BDD representation of the Boolean functions involved is known [1].

Algorithms for BDD construction from a Boolean expression basically follow a recursive scheme according to the above Shannon expansion. Thus they are inherently slow. The same can be said about algorithms for the Boolean operations (and, or, not) on BDD arguments. It should be noted that, given a Boolean function, the size of the resulting BDD is highly dependent on the chosen variable ordering.

3.2 Symbolic representation of LTS

We first define, how elements from finite sets (e.g. set of actions, set of states), are represented as Boolean vectors.

Def: *Encoding*
An *encoding* of a finite set $S = \{s_1, \ldots, s_n\}$ is a mapping $S \rightarrow \{0,1\}^{\lceil \log_2 n \rceil}$. For $x \in S$, we write $enc(x) = (b_1, \ldots, b_{\lceil \log_2 n \rceil})$, i.e. $enc(x)$ is a Boolean vector of length $\lceil \log_2 n \rceil$. ■

The next definition states the way of how to represent LTSs by BDDs.

Def: *Symbolic Representation of a LTS by a BDD*
Let $\mathcal{T} = (S, L, f)$ be a SLTS. For the moment, let us abstract from the rate value of the transition, i.e. let us regard f not as a real-valued function but as a Boolean-valued function $f : S \times L \times S \rightarrow \{0,1\}$ (each non-null rate corresponds to *true*).

Let $\mathcal{B} = (Nodes, Vars, var, low, high)$ be a BDD.
We say that $\mathcal{B}$ is a symbolic representation of $\mathcal{T}$ iff

$$
\begin{aligned}
& x \xrightarrow{a} y \\
\Leftrightarrow & \\
& Bool\_res(Enc(x \xrightarrow{a} y)) = true \\
\Leftrightarrow & \\
& Bool\_res(enc(a), enc(x), enc(y)) = true
\end{aligned}
$$

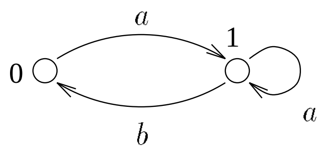
Note that we introduced the function Enc to denote the encoding of the whole of a transition of the LTS, comprising the action label, the source and the target state. ■

Remark: Naturally, we wish to consider "good" variable orderings to achieve "small" BDDs. Let the action label, the source and the target state be encoded by Boolean variables $a_1, \ldots, a_{n_a}$, $s_1, \ldots, s_{n_s}$, and $t_1, \ldots, t_{n_s}$, respectively. Experience has shown that the resulting BDD is small if the ordering of Boolean variables is chosen in the following way [1]:

$$a_1 < \ldots < a_{n_a} < s_1 < t_1 < s_2 < t_2 < \ldots < s_{n_s} < t_{n_s}$$

i.e. the variables encoding the action come first, followed by the variables for source and target state interleaved. In particular, this ordering is advantageous in view of the parallel composition operator discussed below (see sec. 4.1).

Fig. 3 shows a LTS, the way transitions are encoded and the corresponding BDD.



$$
\begin{array}{ccc}
 & & a, s, t \\
0 \xrightarrow{a} 1 & \rightarrow & (0,0,1) \\
1 \xrightarrow{a} 1 & \rightarrow & (0,1,1) \\
1 \xrightarrow{b} 0 & \rightarrow & (1,1,0)
\end{array}
$$

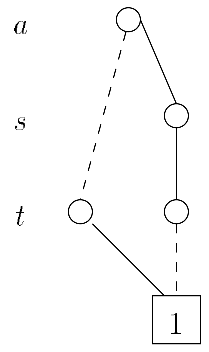


Figure 3: LTS, transition encoding and corresponding BDD

The algorithm for constructing the BDD representation from a given SLTS works as follows: Transitions from the SLTS are processed one by one, each transition being "encoded" in a simple BDD which is subsequently combined by a Boolean "or" operation with the BDD representing all the previously processed transitions. The algorithm can be sketched like this:

```
$T := false$
for each transition $x \xrightarrow{a} y$ of the LTS
    $Newtrans := Enc(x \xrightarrow{a} y)$
    $T := T \vee Newtrans$
```

On the first line, the BDD to be constructed, T, is initialised as $false$. i.e. it does not represent any transition. On the third line, one transition of the SLTS is encoded in BDD $Newtrans := Enc(x \xrightarrow{a} y)$, which has only a single path from the root to the terminal true node, corresponding to action label a and source and target states x and y. On the last line, the "or" between the previous result and the new transition is computed.

4 New data structure: Decision Node BDD

In the previous section we explained the basic idea of representing LTSs symbolically with the help of BDDs. However, we did not specify how to incorporate the rate information into the symbolic representation. Clearly, pure BDDs do not offer any mechanism for representing numerical information.

The basic questions for which we have to find an answer are: How can we map each Boolean assignment $(b_1, \ldots, b_n)$ to a real number, and how can this information be incorporated into the BDD, possibly without changing the basic BDD-structure? In other words, BDDs should be extended in order to represent functions of the type $f : \{0,1\}^n \rightarrow \{0,1\} \times I\!R$. This section explains a new concept to achieve this goal.

We start with the definition of path:

Def: *Path*
A *path* through a BDD is a vector of nodes $(x_1, \ldots, x_k)$, $1 \leq k \leq n+1$, where $x_i \in Nodes$, x_1 is the BDD root node and $x_k \in T$ (x_k is a terminal node) and $\forall i$: $x_{i+1} = low(x_i) \vee x_{i+1} = high(x_i)$.
A path is called a *true-path* iff $x_k = true$, otherwise it is called a *false-path*.
We denote the set of all paths through a BDD by $Paths$.
The set of all true-paths through a BDD is denoted *True-Paths*.
For a given Boolean assignment $(b_1, \ldots, b_n) \in \{0,1\}^n$, the function $path(b_1, \ldots, b_n) = (x_1, \ldots, x_k)$ returns the corresponding path through the BDD.
We define the *length* of a path by $length(x_1, \ldots, x_k) = k$. ■

Remark: If a given path $(x_1, \ldots, x_k)$ has length $k = n+1$, which is the maximal possible length for a path, that path contains a node for every Boolean variable, formally $\forall\ 1 \leq i \leq n : var(x_i) = v_i$. This means that the path corresponds to exactly one Boolean assignment $(b_1, \ldots, b_n)$. In this case, we say that the path does not contain any don't cares. If a path is of length $k < n+1$, it contains $n+1-k = d$ don't cares.

Such a path corresponds to 2^d different Boolean assignments (because for every don't care two Boolean assignments are possible).

Every Boolean assignment is mapped onto exactly one path. Several Boolean assignments (always a power of 2) may be mapped onto the same path, in which case the path has one or more don't cares. Therefore we assign to each path a real-valued vector, a vector of rates, also called a rate list, whose length (a power of 2) is determined by the number of don't cares of the path. Formally, we introduce the function $rates(x_1, \ldots, x_k) = (r_1, \ldots, r_{2^{n+1-k}})$. Thus, the correspondence of Boolean assignments to rates is one to one, uniquely defined by the lexical ordering of the Boolean assigments. We illustrate this concept in fig. 4.

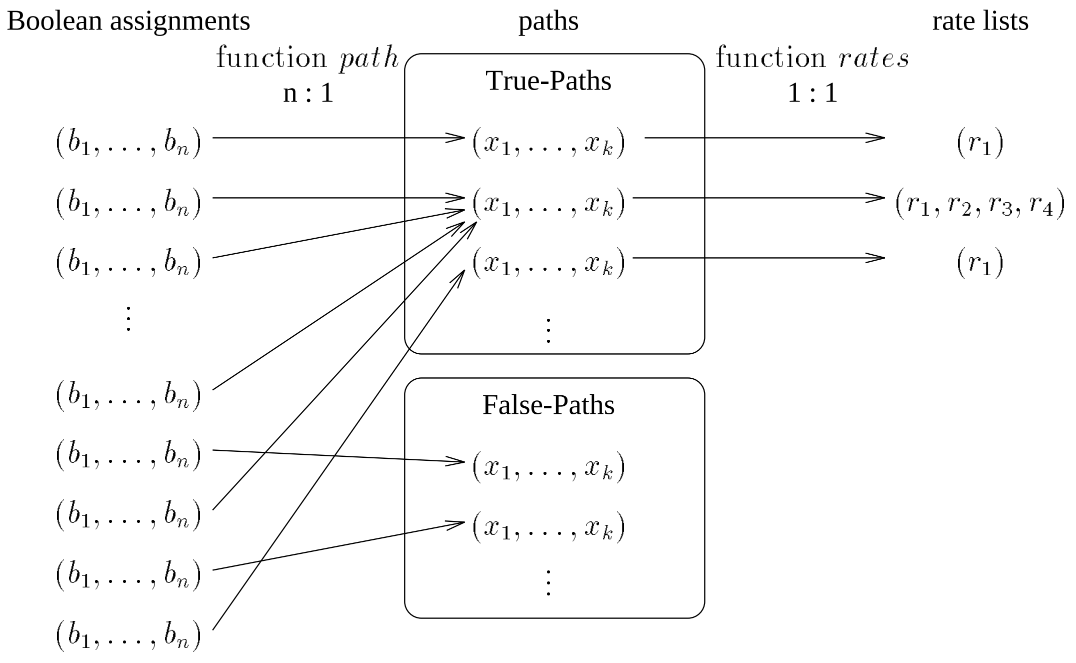


Figure 4: correspondence between Boolean assignments, paths and rate lists

We can now give the central definition for our new data structure. Note that the name, Decision Node BDD, will become clear from the discussion below about the practical realisation of the concept.

Def: *Decision Node BDD*
A *Decision Node BDD* (DNBDD) is a BDD extended by a function

$$rates : \textit{True-Paths} \rightarrow \mathbb{R}^{2^{n+1-k}}$$

i.e. a real-valued vector is assigned to every true-path. The length of the vector depends on the length of the true-path ($k = \textit{length}(\textit{true-path})$ is not a global constant but depends on the individual true-path). ■

So far, we decided that every true-path is mapped onto a real-valued vector whose dimension is given by the number of Boolean assignments corresponding to the path. Next we must find a practical method for attaching that information to the BDD. What are the characteristics of a path? A subset of the BDD nodes, the so-called *Decision Nodes* play a key role in this consideration.

Def: *Decision Node*
A non-terminal BDD-node $x \in NT$ is called *decision node* iff $low(x) \neq false \wedge high(x) \neq false$, i.e. iff the terminal true-node can be reached via both outgoing edges of node x. The set of decision nodes is denoted DN. ■

Let $(x_1, \ldots, x_k) \in$ *True-Paths*. Let x_j be the "last" decision node on that path, i.e. $x_j \in DN \wedge \forall\ j < l \leq k : x_l \notin DN$. We then attach the rate-vector $rates(x_1, \ldots, x_k) = (r_1, \ldots, r_{2^{n+1-k}})$ to the edge (x_j, x_{j+1}). This concept is illustrated in fig. 5 (in the figure, decision nodes are drawn black). In this example, there are four Boolean assignments evaluating to *true*, each of which is mapped onto a rate as shown in the left part of the figure. The first two assignments are mapped onto the same path, a path which has a don't care in the Boolean variable s. Therefore, the corresponding rate list (λ, μ) has length two.

(a,s,t)	$\rightarrow$	*rate*
$(0,0,1)$	$\rightarrow$	λ
$(0,1,1)$	$\rightarrow$	μ
$(1,0,1)$	$\rightarrow$	α
$(1,1,0)$	$\rightarrow$	β

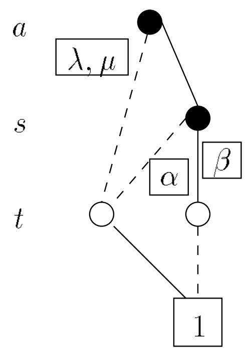


Figure 5: mapping of Boolean assignments to rates and corresponding DNBDD

The practical realisation of the DNBDD concept introduced so far induces the following problem: There are situations, where several true-paths share their last decision node. This is the case if and only if there exists a decision node which can be reached from the root by more than one path. In such a case, several rate lists would be assigned to the same edge. This would result in a confusion, since it would not be clear any more which rate list corresponds to which true-path. As an example, see fig. 6 (left), where a decision node has more than one incoming edge. In order to overcome this problem, we introduce a pointer structure as illustrated in fig. 6 (right). We refer to this pointer structure as the *rate tree* of a DNBDD. Using rate trees in the practical realisation of DNBDDs, the mapping from true-paths to rate lists is one-to-one, as it should be.

In addition to the function *Bool_res*, which can remain unchanged as defined before,

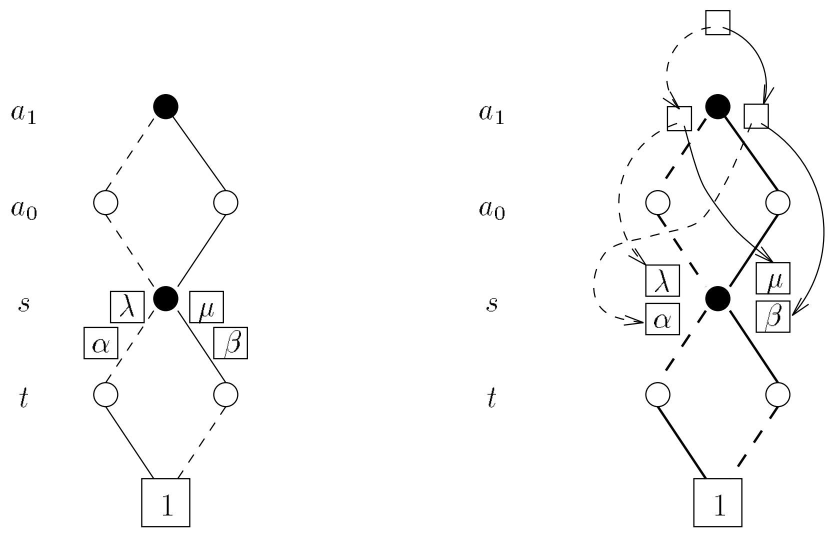


Figure 6: two true-paths sharing their last decision node, DNBDD with rate tree

we now define a function *Num_res*, which, given a Boolean assignment, computes the numerical result.

Def: *Numeric result value* $Num\_res(b_1, \ldots, b_n)$
Let $(b_1, \ldots, b_n)$ be a fixed assignment to the Boolean variables $v_1, \ldots, v_n$.
If $Bool\_res(b_1, \ldots, b_n) = false$ then the function $Num\_res(b_1, \ldots, b_n)$ is undefined.
Else let $path(b_1, \ldots, b_n) = (x_1, \ldots, x_k)$ and $rates(x_1, \ldots, x_k) = (r_1, \ldots, r_{2^{n+1-k}})$. Then $Num\_res(b_1, \ldots, b_n) = r_i$ where i is determined unambiguously by those positions of $(b_1, \ldots, b_n)$ which correspond to don't cares. In other words, each of the 2^{n+1-k} Boolean assignments sharing path $(x_1, \ldots, x_k)$ corresponds to exactly one element of the rate list $(r_1, \ldots, r_{2^{n+1-k}})$, and this correspondence is according to the lexicographical ordering of the Boolean assignments.

We are now able to define how to represent a SLTS by a DNBDD:

Def: *Symbolic Representation of a SLTS by a DNBDD*
Let $\mathcal{T} = (S, L, f)$ be a SLTS.
Let $\mathcal{B} = (Nodes, Vars, var, low, high, rates)$ be a DNBDD.
We say that $\mathcal{B}$ is a symbolic representation of $\mathcal{T}$ iff

$$x \xrightarrow{a,\lambda} y$$
$$\Leftrightarrow$$
$$(Bool\_res(Enc(x \xrightarrow{a} y)) = true) \wedge (Num\_res(Enc(x \xrightarrow{a} y)) = \lambda)$$
$$\Leftrightarrow$$
$$(Bool\_res(enc(a), enc(x), enc(y)) = true) \wedge (Num\_res(enc(a), enc(x), enc(y)) = \lambda)$$

■

Fig. 7 shows two example SLTSs and their DNBDD representation. Note that the first example is the same as the one given in fig. 3, augmented by the rate information. In the second example, there are four different actions which are encoded in two bits (a_1 and a_0). This example has a slightly more complex rate tree.

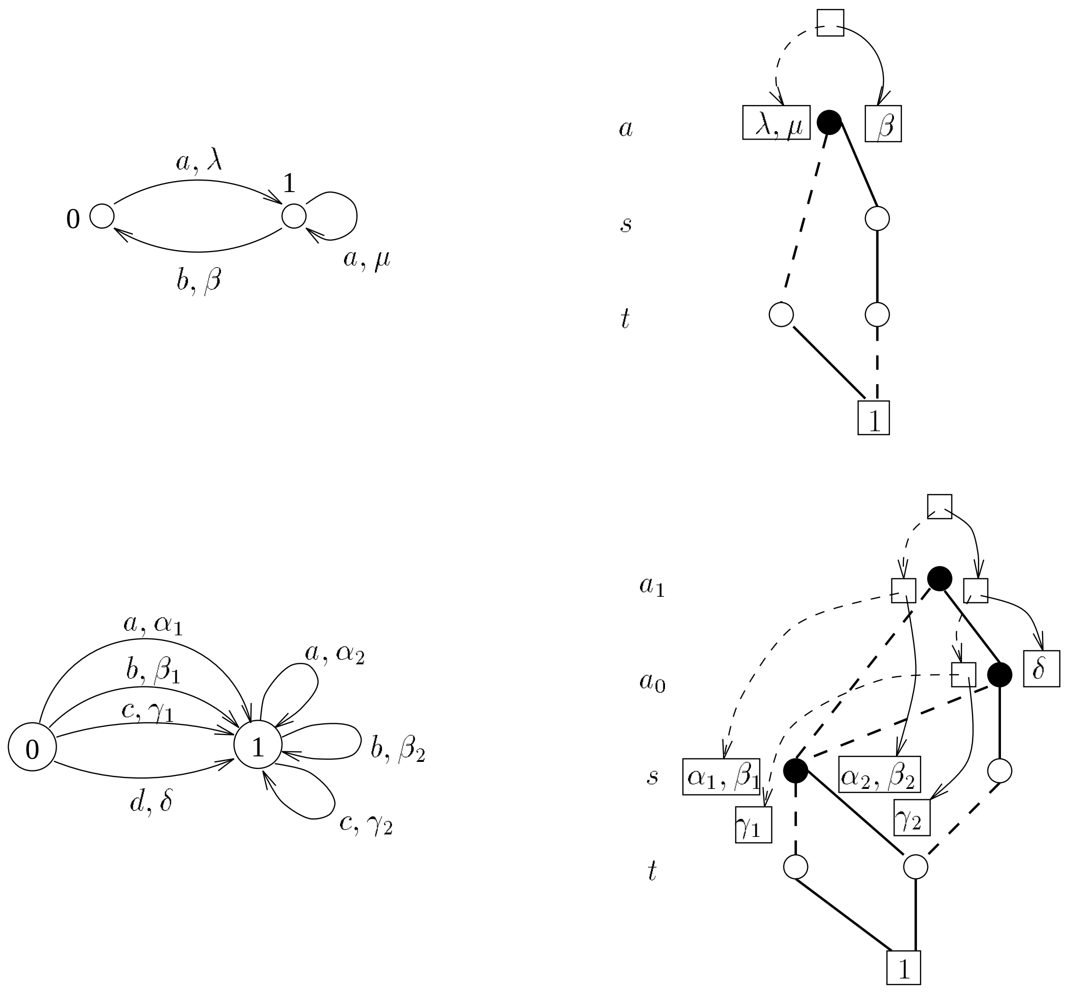


Figure 7: SLTS and corresponding DNBDD

4.1 Operations on DNBDD

The method of generation of a DNBDD from a given STLS is basically the same as explained earlier for the purely functional case (see sec. 3.2), i.e. transitions are processed one by one. Each transition is first translated into a very simple DNBDD which is then combined by an or-operation with the previously obtained intermediate result. Of course, the or-operation used in this procedure has to be capable of building and manipulating the rate-tree.

For DNBDDs representing LTSs which originate from stochastic process algebras, an important operation is parallel composition. The parallel composition operator of the SPA can be realised directly on the DNBDD representation of the two operand processes. Suppose we wish to perform the parallel composition of two processes, $P = A \parallel_S B$, where S denotes the set of synchronising actions, i.e. those actions which both partners perform simultaneously together. We assume that the DNBDDs which correspond to processes A and B have already been generated and are denoted $\mathcal{A}$ and $\mathcal{B}$. The set S can also be coded as a BDD, namely $\mathcal{S}$ (note that $\mathcal{S}$ is a BDD and not a DNBDD, since it does not contain any rate information). The DNBDD $\mathcal{P}$ which corresponds to the resulting process P can then be written as a Boolean expression:

$$\begin{array}{rcl} \mathcal{P} & = & (\mathcal{A} \wedge \mathcal{S}) \wedge (\mathcal{B} \wedge \mathcal{S}) \\ & \vee & (\mathcal{A} \wedge \overline{\mathcal{S}} \wedge Stab_B) \\ & \vee & (\mathcal{B} \wedge \overline{\mathcal{S}} \wedge Stab_A) \end{array}$$

The term on the first line is for the synchronising actions in which both A and B participate. The term on the second (third) line is for those actions which A (B) performs independently of B (A) — these actions are all from the complement of S. The meaning of $Stab_A$ ($Stab_B$) is a BDD which expresses stability. i.e. the fact that the source state of process A (B) equals its target state.

An important question is about the result rate of synchronising actions. Depending on the application, different expressions for the result rate may apply. Typical examples are the maximum, minimum, sum or product of the two partner rates. If the product of the two partner rates is chosen, the concept of compositionality is supported [19]. Using DNBDDs, the result rate will be calculated from the two partner rates during the and-operation at the center of the first line of the above equation. This and-operation is flexible enough to realise any of the above alternatives (maximum, minimum, ...), i.e. DNBDDs cover any of those cases.

The result, $\mathcal{P}$, describes all transitions which are possible in the product space of the two processes. Given a pair of initial states for A and B, only part of the product space may be reachable due to synchronisation conditions. Reachability analysis can be performed on the DNBDD representation, restricting $\mathcal{P}$ to those transitions which originate in reachable states.

4.2 Symbolic minimisation of SLTS, working on DNBDD

This subsection describes how a SLTS can be minimised based on an equivalence relation defined on the set of states. The idea is to reduce the state space by representing all equivalent states by a single macro state. It is shown how such a minimisation technique can be applied to the DNBDD representation of the SLTS, i.e. the minimisation is entirely based on DNBDD operations. Symbolic minimisation based on BDDs for the purely functional case has been described before, see e.g. [2]. To the best of our knowledge, BDD-based minimisation of *stochastic* LTSs, i.e. BDD-based minimisation which takes into account the stochastic rate information, is a new approach. The advantages for performance analysis are obvious: The stochastic LTSs of a complex system can be built from small components by applying the DNBDD-based parallel composition operator step by step. After every parallel composition step, the intermediate result can be minimised without leaving the DNBDD world. Thus, the use of DNBDDs quite ideally supports the concept of compositional reduction.

The equivalence relation on which we focus is known as Markovian bisimulation [19]. Informally, two states are Markov-bisimilar (members of the same equivalence class) iff from both states all equivalence classes can be reached in one step by the same actions and with the same cumulative rate (defined below). There is a strong connection between Markovian bisimulation and classical Markov chain *lumpability* [22].

We start by defining the notion of partition:

Def: *Partition*
Let S be a finite set. A family of subsets (also called "classes") $C_i, 1 \leq i \leq n$, with $C_i \subseteq S$, is called *partition* of S, iff

- $C_i \cap C_j = \emptyset$ for $i \neq j$, and
- $\bigcup_{1 \leq i \leq n} = S$ ■

Fig. 8 illustrates how the state space S is partitioned into three disjoint subsets.

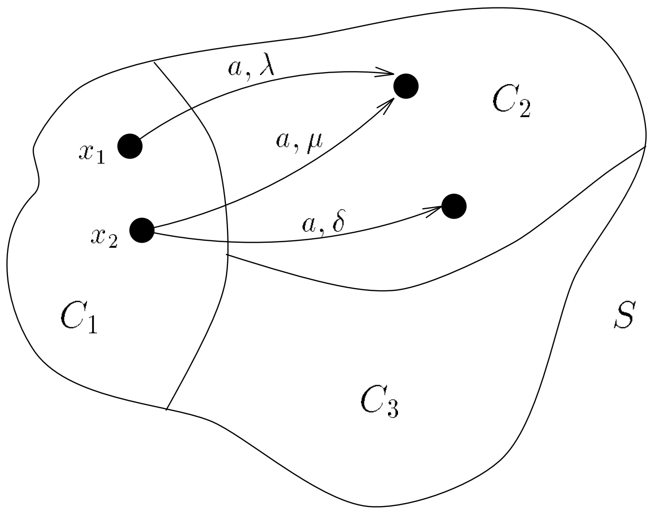


Figure 8: Partitioning of state space S

Next, we need to define the concept of cumulative rate:

Def: *Cumulative Rate*
Let $C_1, \ldots, C_n$ be a partition of the state space S of a SLTS. Let $x \in S$. The *cumulative rate* of action a from state x to class i is defined as

$$\Lambda(x, a, i) = \sum_{x \xrightarrow{a,\lambda} y,\; y \in C_i} \lambda$$

■

For example, in fig. 8, $\Lambda(x_1, a, 2) = \lambda$ and $\Lambda(x_2, a, 2) = \mu + \delta$.

Remark: When using DNBDDs, the cumulative rate of action a_k from state x to class i can be easily computed in the following way. Let $T(a, s, t)$ be the DNBDD representing a SLTS (a, s and t are vectors of Boolean variables). For convenience, T is usually broken up into individual DNBDDs $T_{a_k}(s, t)$, one for every action a_k:

$$T_{a_k}(s, t) = (T(a, s, t) \wedge (a = a_k))$$

In order to obtain a DNBDD which represents all transitions from state x to states from class i we restrict $T_{a_k}(s, t)$ to the single source state x and to target states from class i (class i is represented by a BDD $C_i(t)$):

$$T_{x \xrightarrow{a_k} C_i}(s, t) = (T_{a_k}(s, t) \wedge (s = x) \wedge C_i(t))$$

The cumulative rate is then computed by applying the function *soar* (sum of all rates) to $T_{x \xrightarrow{a_k} C_i}(s, t)$. This function simply sums up all the entries in the rate tree of a DNBDD.

We can now give the formal definition of Markovian bisimulation.

Def: *Markovian Bisimulation*
Let $C_1, \ldots, C_n$ be a partition of the state space S of a SLTS. Let $\overset{M}{\sim}$ be the equivalence relation corresponding to this partition. $\overset{M}{\sim}$ is called a *Markov Bisimulation* iff

$$\forall x_1, x_2 \in S : x_1 \overset{M}{\sim} x_2 \Rightarrow \forall a : \forall C_i : \Lambda(x_1, a, i) = \Lambda(x_2, a, i)$$

■

Algorithms for Markovian bisimulation traditionally follow an iterative refinement scheme [24, 11, 21]. For instance, an implementation in the context of the TIPP tool [17, 18] is decribed in [12]. This means that starting from an initial partition which consists of a single class (containing all states), classes are refined until the obtained partition corresponds to a Markovian bisimulation. The result thus obtained is the largest existing Markovian bisimulation, in a sense the "best" such bisimulation, since it has a minimal number of equivalence classes.

For the refinement of a partition, the notion of a "splitter" is very important. A splitter is a pair (a, C_{spl}), consisting of an action a and a class C_{spl}. During refinement, a class C_i is split with respect to a splitter, which means that subclasses $C_{i1}, C_{i2}, \ldots, C_{ik}$ are computed ($k \geq 1$), such that the cumulative rate $\Lambda(x, a, spl)$ is the same for all the states x belonging to the same subclass.

In the following, a DNBDD-based bisimulation algorithm is presented, in which the transition system is represented by DNBDDs $T_a(s,t)$, one for each action a, and in which the current partition is stored as a set of BDDs, one for each class. The algorithm uses a dynamic set of splitters, denoted *Splitters*, which can be realised as a pointer structure. Note that here we only present a basic version of the algorithm which can be optimised in many ways [7, 15].

```
1. Initialisation
   $Partition := \{C_1\} = \{S\}$
   /* the initial partition consists of only one class which contains all states */
   $Splitters := Act \times C_1$
   /* all pairs of actions and classes have to be considered as splitters*/

2. Main loop

   while ($Splitters \neq \emptyset$)
       choose splitter $(a, C_{spl})$
       forall $C_i$  $split(C_i, a, C_{spl})$
       /* all classes (including $C_{spl}$ itself) are split */
       $Splitters := Splitters - (a, C_{spl})$
       /* the processed splitter is removed from the splitter set */
```

It remains to specify the procedure *split*. Its task is to split a class C_i, using the combination (a, C_{spl}) as a splitter. Procedure *split* uses a data structure *split_tree* which is shown in fig. 9. The input class C_i is split into subclasses $C_{i1}, \ldots, C_{ik}$ according to the cumulative rate from a state in C_i to class C_{spl} (regarding transitions labelled with action a). The subclasses $C_{i1}, \ldots, C_{ik}$ are represented by BDDs.

```
procedure $split(C_i, a, C_{spl})$
    forall $s_x \in C_i$
        $\Lambda_{s_x} = soar(T_a(s,t) \wedge (s = s_x) \wedge C_{spl}(t))$
        /* the cumulative rate from state $s_x$ to $C_{spl}$ is computed */
        $insert(split\_tree, s_x, \Lambda_{s_x})$
        /* state $s_x$ is inserted into the split_tree */
    if ($k > 1$) /* if $C_i$ has been split into $k > 1$ subclasses */
        $Partition := Partition \cup \{C_{i1}, C_{i2}, \ldots, C_{ik}\} - C_i$
        $Splitters := Splitters \cup (Act \times \{C_{i1}, C_{i2}, \ldots, C_{ik}\}) - Act \times C_i$
        /* the partition and the splitter set are updated */
```

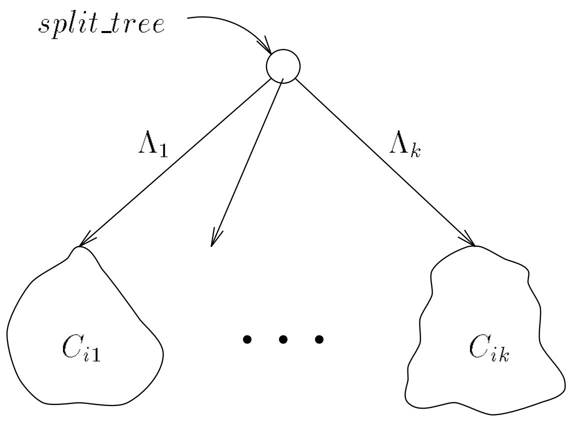


Figure 9: *split_tree* used by procedure *split*

Remark: In the forall loop of procedure *split*, the cumulative rate is computed for every state s_x in class C_i, and state s_x is inserted into the *split_tree* accordingly. If splitting has taken place, the partition must be refined and the set of splitters must be updated.

5 Tool

The concept of DNBDDs which we introduced in this paper has been fully realised in a tool [3], whereby the feasibility of our approach has been proved. The tool is still a prototype which is not highly optimised in terms of memory requirements and efficiency (our human resources were quite limited). Therefore, so far, the tool is not capable of handling very large state spaces. The tool is written in C and up to now only offers a rudimentary textual user interface. Its main capabilities are:

- generation of a DNBDD from a given SLTS. Currently, the tool reads SLTS files generated by the TIPP tool [17, 18], a tool for the specification and analysis of SPA models.
- parallel composition of two processes whose behaviour had been encoded into DNBDD form in previous steps. Afterwards, if the user desires, reachability analysis can be performed in order to restrict the potential behaviour of the resulting process to the states which are actually reachable.
- minimisation on the basis of Markovian bisimulation. We implemented a version of the bisimulation algorithm which uses the splitter set administration technique explained earlier (see sec. 4.2). As a result, the tool outputs the final partition of the state space and generates the DNBDD representation of the reduced transition system.

- all results computed by the tool in DNBDD form can be converted back into their SLTS representation.

6 Conclusion

The problem of state space explosion remains the most serious problem of analytical performance and dependability modelling. A lot of research has been done on how to best avoid or tolerate large state spaces, some of which produced very valuable results. Nevertheless it remains important to look out for new ideas which may improve the tractability of complex models. In this sense, symbolic techniques, in particular BDDs, are very promising, since they have been highly successfully used in state-space-based techniques in the area of functional analysis. It must be noted that, in the past, the symbolic approach had not received much consideration from the performance community, which seems to make it all the more important to be looked into now.

We developed DNBDDs, a new data structure which is an extension of basic BDDs, tailored to represent *stochastic* transition systems in a compact way. We were able to show that all the algorithms which are generally needed to build, manipulate and analyse SLTSs have a corresponding algorithm which works on the more compact DNBDD representation. Using our new technique, the advantages of the symbolic approach can be enjoyed not only while working on purely functional considerations, but also during analyses which make use of the numerical information of a SLTS. Therefore the use of DNBDDs has the potential to handle more complex stochastic performance models than before.

Acknowledgements: The author had many fruitful discussions with Holger Hermanns, in particular on the subject of bisimulation. Hannes Bruchner, in addition to implementing the tool as part of a student's project, contributed many helpful ideas.

References

[1] H.R. Andersen. An Introduction to Binary Decision Diagrams. Technical report, Department of Computer Science, Technical University of Denmark, December 1994.

[2] A. Bouali and R. de Simone. Symbolic Bisimulation Minimisation. In *Computer Aided Verification*, pages 96–108, 1992. LNCS 663.

[3] H. Bruchner. Symbolische Manipulation von stochastischen Transitionssystemen. Internal study, Universität Erlangen–Nürnberg, IMMD VII, 1998. in German.

[4] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE ToCS*, C-35(8):677–691, August 1986.

[5] R.E. Bryant. Symbolic Boolean Manipulation woth Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] R.E. Bryant and Y. Chen. Verification of Arithmetic Functions with Binary Moment Diagrams. Technical Report CMU-CS-94-160, CMU, 1994.

[7] P. Buchholz. A Framework for the Hierarchical Analysis of Discrete Event Dynamic Systems. Habilitation thesis, Universität Dortmund, 1996.

[8] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, (98):142–170, 1992.

[9] E.M. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multiterminal Binary Decision Diagrams: An efficient data structure for matrix representation. In *IWLS: International Workshop on Logic Synthesis*, Tahoe City, May 1993.

[10] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, (6):155–164, 1993.

[11] J.C. Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13:219–236, 1989.

[12] R. Foldenauer. Implementierung von Algorithmen zur Äquivalenzüberprüfung in das TIPPtool. Internal study, Universität Erlangen–Nürnberg, IMMD VII, Dezember 1996. in German.

[13] N. Götz, H. Hermanns, U. Herzog, V. Mertsiotakis, and M. Rettelbach. Stochastic Process Algebras – Constructive Specification Techniques Integrating Functional, Performance and Dependability Aspects. In F. Baccelli, A.J. Marie, and I. Mitrani, editors, *Quantitative Methods in Parallel Systems*. Springer, 1995.

[14] G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian Analysis of Large Finite State Machines. *IEEE Trans. on CAD*, 15(12):1479–1493, Dec. 1996.

[15] H. Hermanns. *Interactive Markov Chains*. PhD thesis, Universität Erlangen-Nürnberg, 1998. to appear.

[16] H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic Process Algebras as a Tool for Performance and Dependability Modelling. In *Proc. of IEEE International Computer Performance and Dependability Symposium*, pages 102–111, Erlangen, April 1995. IEEE Computer Society Press.

[17] H. Hermanns and V. Mertsiotakis. A Stochastic Process Algebra Based Modelling Tool. In M. Merabti, M. Carew, and F. Ball, editors, *Performance Engineering of Computer and Telecommunications Systems*, pages 187–201. Springer, 1996.

[18] H. Hermanns, V. Mertsiotakis, and M. Rettelbach. A Construction and Analysis Tool Based on the Stochastic Process Algebra TIPP. In *Proc. of 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 427–430. Springer, LNCS 1055, 1996.

[19] H. Hermanns and M. Rettelbach. Syntax, Semantics, Equivalences, and Axioms for MTIPP. In U. Herzog and M. Rettelbach, editors, *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 71–88, Erlangen-Regensberg, July 1994. IMMD, Universität Erlangen-Nürnberg.

[20] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[21] P. Kanellakis and S. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

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

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

[24] R. Paige and R. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[25] M. Rettelbach and M. Siegle. Compositional Minimal Semantics for the Stochastic Process Algebra TIPP. In U. Herzog and M. Rettelbach, editors, *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 89–106, Regensberg/Erlangen, July 1994. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg 27 (4).

[26] M. Siegle. Structured Markovian Performance Modelling with Automatic Symmetry Exploitation. In G. Haring and H. Wabnig, editors, *Short Papers and Tool Descriptions Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 77–81, Vienna, Austria, May 1994.

[27] M. Siegle. BDD extensions for stochastic transition systems. In D. Kouvatsos, editor, *Proc. of 13th UK Performance Evaluation Workshop*, pages 9/1 – 9/7, Ilkley/West Yorkshire, July 1997.