Symbolic Minimisation of Stochastic Process Algebra Models

Holger Hermanns[1] and Markus Siegle[2]

[1]Systems Validation Centre, FMG/CTIT, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: hermanns@cs.utwente.nl

[2]Informatik 7, University of Erlangen-Nürnberg,
Martensstraße 3, 91058 Erlangen, Germany
e-mail: siegle@informatik.uni-erlangen.de

Abstract: Stochastic process algebras have been introduced in order to enable compositional performance analysis. The size of the state space is a limiting factor, especially if the system consists of many cooperating components. To fight state space explosion, compositional aggregation based on congruence relations can be applied. This paper addresses the computational complexity of minimisation algorithms and explains how efficient, BDD-based data structures can be employed for this purpose.

1 Introduction

Stochastic Process Algebras (SPA) have been developed as a formal description technique for the specification and design of distributed systems. A gentle introduction to SPAs is provided in [9]. In addition to classical process algebras, SPAs incorporate information about a system's temporal behaviour, thereby enabling the modelling of performance and reliability aspects. We consider SPAs where the delay of an action is either exponentially distributed (Markovian actions) or equal to zero (immediate actions). Similar to other specification techniques, the phenomenon of state space explosion can be frequently observed when working with SPA specifications. However, the algebraic foundations of SPAs and their concept of compositionality enable efficient techniques for state space reduction.

Compositional modelling of distributed systems with SPAs is particularly successful if the system structure can be exploited during Markov chain generation. For this purpose, congruence relations have been developed which justify minimisation of components without touching behavioural properties. Minimised components can be plugged into the original model in order to circumvent the state space explosion problem. This strategy, known as *compositional aggregation* has been applied successfully to handle very complex models (see, e.g. [11]).

Applicability of compositional aggregation relies on the existence of *algorithms* to compute minimised components. We discuss efficient algorithms for strong equivalence, and (strong and weak) Markovian bisimulation. The algorithms are variants of well-known partition refinement algorithms [19, 6, 15]. They compute partitions of equivalent states of a given state space by iterative refinement of partitions, until a fixed point is reached.

For the compact representation of SPA models and for the practical realisation of the algorithms we introduce data structures based on Binary Decision Diagrams (BDDs) [2]. During the recent years, BDDs have established themselves as the state-of-the-art in such areas as digital circuit verification and model checking. The success of BDDs is due to the fact that they enable an efficient, *symbolic* encoding of state spaces. In the context of SPAs, it is particularly important that parallel composition of components can be realised on the BDD data structure in a way which avoids the usually observed exponential blow-up due to the interleaving of causally independent transitions [5]!

Pure BDDs are not capable of representing the information on transition rates which is part of an SPA specification. For that purpose, we have developed a stochastic extension of BDDs which we call Decision Node BDDs (DNBDDs). A DNBDD is a BDD decorated with additional numerical information. DNBDDs therefore make it possible to incorporate information about transition

rates into the symbolic representation. Unlike other numerical extensions of BDDs, DNBDDs do not alter the basic BDD structure. We highlight how parallel composition and compositional aggregation can both be performed efficiently in a stochastic setting with the help of DNBDDs.

This paper is organised as follows: Sec. 2 contains the definition of the languages and of the bisimulation relations which we consider. Sec. 3 and Sec. 4 present the basic bisimulation algorithms for non-stochastic process algebras and for the purely Markovian case. In Sec. 5, we focus on BDDs and DNBDDs and show how algorithms for parallel composition and bisimulation can benefit from the use of these data structures. Sec. 6 contains the conclusion. This paper is an abridged version of [12].

2 Basic definitions

In this section we define the SPA language and its operational semantics. In addition, we recall the notions of strong and weak Markovian bisimilarity. We refer to [9] for more details.

Definition 1 *Let Act be the set of valid action names and Pro the set of process names. We distinguish the action* $\mathtt{i}$ *as an internal, invisible activity. Let $a \in Act$, $P,P_i \in \mathcal{L}$, $A \subseteq Act \setminus \{\mathtt{i}\}$, and $X \in Pro$. The set $\mathcal{L}$ of expressions consists of the following language elements:*

$\mathtt{stop}$	inaction		
$a\ ;\ P$	action prefix	$(a,\lambda)\ ;\ P$	Markovian prefix
$P_1\ []\ P_2$	choice	$P_1\ \|[A]\|\ P_2$	parallel composition
$\mathtt{hide}\ a\ \mathtt{in}\ P$	hiding	X	process instantiation

A set of process definitions (of the form $X := P$) constitutes a process environment.

A set of operational semantic rules [12] defines a labelled transition system (LTS) containing action transitions, $\xrightarrow{a}$, and Markovian transitions, $\xrightarrow{a,\lambda}$. For synchronisation of Markovian transitions, a function ϕ determines the rate of synchronisation, since different synchronisation policies (minimum, maximum, product, ... of the two partner rates) are possible.

Strong and weak Markovian bisimilarity are defined using the function $\gamma : \mathcal{L} \times Act \times 2^{\mathcal{L}} \mapsto \mathbb{R}$, often called the *cumulative rate*, defined as follows (we use $\{\!|$ and $|\!\}$ to denote multiset brackets):

$$\gamma(P,a,C) := \textstyle\sum_{\lambda \in E(P,a,C)} \lambda, \text{ where } E(P,a,C) := \{\!|\ \lambda \mid P \xrightarrow{a,\lambda} P' \wedge P' \in C\ |\!\}.$$

Definition 2 *An equivalence relation $\mathcal{B}$ is a strong Markovian bisimulation, if $(P,Q) \in \mathcal{B}$ implies*
(i) $P \xrightarrow{a} P'$ implies $Q \xrightarrow{a} Q'$, for some Q' with $(P',Q') \in \mathcal{B}$,
(ii) for all equivalence classes C of $\mathcal{B}$ and all actions a it holds that

$$\gamma(P,a,C) = \gamma(Q,a,C).$$

Two expressions P and Q are strong Markovian bisimilar (written $P \sim Q$) if they are contained in a strong Markovian bisimulation.

Weak bisimilarity is obtained from strong bisimilarity by basically replacing $\xrightarrow{a}$ with $\stackrel{a}{\Longrightarrow}$. Here, $\stackrel{a}{\Longrightarrow}$ denotes an observable a transition that is preceded and followed by an arbitrary number (including zero) of invisible activities, i.e. $\stackrel{a}{\Longrightarrow} := \xrightarrow{\mathtt{i}^*} \xrightarrow{a} \xrightarrow{\mathtt{i}^*}$. As discussed in [9], the extension from strong to weak Markovian bisimilarity has to take into account the interplay of Markovian and immediate transitions. Priority of *internal* immediate transitions gives rise to the following definition.

Definition 3 *An equivalence relation $\mathcal{B}$ is a weak Markovian bisimulation, if $(P,Q) \in \mathcal{B}$ implies*
(i) $P \stackrel{a}{\Longrightarrow} P'$ implies $Q \stackrel{a}{\Longrightarrow} Q'$, for some Q' with $(P',Q') \in \mathcal{B}$,
(ii) if $P \stackrel{\mathtt{i}}{\Longrightarrow} P' \not\xrightarrow{\mathtt{i}}$ then there exists Q' such that $Q \stackrel{\mathtt{i}}{\Longrightarrow} Q' \not\xrightarrow{\mathtt{i}}$, and
for all equivalence classes C of $\mathcal{B}$ and all actions a

$$\gamma(P',a,C) = \gamma(Q',a,C).$$

Two expressions P and Q are weak Markovian bisimilar (written $P \approx Q$) if they are contained in a weak Markovian bisimulation.

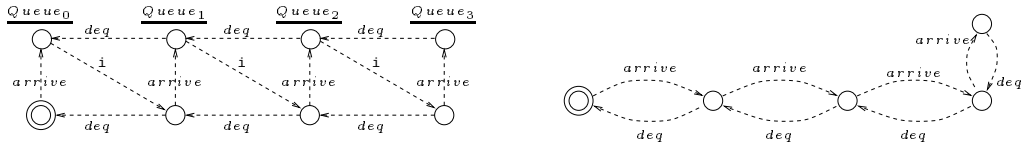

Figure 1: LTS of the queueing system example, before and after applying weak bisimilarity

In this definition, $P \not\xrightarrow{\mathtt{i}}$ denotes that P does not possess an outgoing internal immediate transition. We call such a state a *tangible* state, as opposed to *vanishing* states which may internally and immediately evolve to another behaviour (denoted $P \xrightarrow{\mathtt{i}}$).

It can be shown that strong Markovian bisimilarity is a congruence with respect to the language operators, provided that ϕ is distributive over summation of real values. The same result holds for weak Markovian bisimilarity except for congruence with respect to choice, see [8].

In the sequel, we consider two distinct sub-languages of $\mathcal{L}$. The first, $\mathcal{L}_1$, arises by disallowing Markovian prefix, resulting in a non-stochastic process algebra, a subset of Basic LOTOS [1], where only action transitions appear in the underlying LTS. On this language, strong and weak Markovian bisimilarity coincide with Milner's non-stochastic strong and weak bisimilarity [18]. The complementary subset, $\mathcal{L}_2$, is obtained by disallowing the other prefix, action prefix. The resulting language coincides with MTIPP à la [10] (if ϕ is instantiated with multiplication), and both strong and weak Markovian bisimilarity coincide with Markovian bisimilarity on MTIPP. Note that Markovian bisimilarity agrees with Hillston's strong equivalence [14]. The semantics of $\mathcal{L}_2$ only contains Markovian transitions, and we will refer to such a transition system as a stochastic LTS (SLTS). For a treatment of the complete language where both prefixes coexist, we refer the reader to [12].

3 The non-stochastic case

In this section, we introduce the general idea of iterative partition refinement, working with the language $\mathcal{L}_1$. To illustrate the key ideas, we use as an example a queueing system, consisting of an arrival process and a finite queue. First, we model an arrival process as an infinite sequence of incoming arrivals ($arrive$), each followed by an enqueue action (enq).

$$Arrival := arrive;\ enq;\ Arrival$$

The behaviour of the queue is described by a family of processes, one for each value of the current queue population.

$$\begin{aligned} Queue_0 &:= enq;\ Queue_1 \\ Queue_i &:= enq;\ Queue_{i+1}\ []\ deq;\ Queue_{i-1} \qquad 1 \leq i < max \\ Queue_{max} &:= deq;\ Queue_{max-1} \end{aligned}$$

These separate processes are combined by parallel composition in order to describe the whole queueing system. Hiding is used to internalise actions as soon as they are irrelevant for further synchronisation.

$$System := \mathbf{hide}\ enq\ \mathbf{in}\ \left(Arrival\ |[enq]|\ Queue_0\right)$$

Fig. 1 (left) shows the LTS associated with the $System$ specified above for the case that the maximum queue population is $max = 3$. The LTS has 8 states, the initial state being emphasised by a double circle. Fig 1 (right) shows an equivalent representation, minimised with respect to weak bisimilarity. The original state space is reduced by replacing every class of weakly bisimilar states by a single state.

Most algorithms for computing bisimilarity require a *finite* state space and follow an iterative refinement scheme [19, 6, 15]. This means that starting from an initial partition of the state space which consists of a single class (containing all states), classes are refined until the obtained partition corresponds to a bisimulation equivalence. The result thus obtained is the largest existing 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 is split with respect to a splitter, which means that subclasses C^+ and C^- are computed, such that subclass C^+ contains all those states from C which can perform an a-transition leading to class C_{spl}, and C^- contains all remaining states. In the following, an algorithm for strong bisimulation is presented which uses a dynamic set of splitters, denoted $Splitters$. Note that here we only present a basic version of the algorithm which can be optimised in many ways [6, 19]. By a deliberate treatment of splitters, it is possible to obtain a time complexity $\mathcal{O}(m \log n)$, where n is the number of states and m is the number of transitions [8].

1. **Initialisation**
   $Partition := \{S\}$
   /* the initial partition consists of only one class which contains *all* states */
   $Splitters := Act \times Partition$
   /* all pairs of actions and classes have to be considered as splitters */
2. **Main loop**
   while $(Splitters \neq \emptyset)$
   choose splitter $(a, C_{spl}) \in Splitters$
   forall $C \in Partition$ $\quad split(C, a, C_{spl}, Partition, Splitters)$
   /* 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, using (a, C_{spl}) as a splitter. If splitting actually takes place, the input class C is split into subclasses C^+ and C^-.

procedure $split(C, a, C_{spl}, Partition, Splitters)$
$C^+ := \{P \mid P \in C \wedge \exists\, Q : (P \xrightarrow{a} Q \wedge Q \in C_{spl})\}$ /* the subclass C^+ is computed */
if $(C^+ \neq C \wedge C^+ \neq \emptyset)$ /* only continue if class C actually needs to be split */
$C^- := C - C^+$ /* C^- is the complement of C^+ with respect to C */
$Partition := Partition \cup \{C^+, C^-\} - \{C\}$
$Splitters := Splitters \cup (Act \times \{C^+, C^-\}) - Act \times \{C\}$
/* the partition and the splitter set are updated */

4 The Markovian case

In this section, we consider the language $\mathcal{L}_2$ where *all* actions are associated with an exponentially distributed delay. In addition, ϕ is instantiated with the product of rates, for reasons discussed (for instance) in [9]. The semantic model of a process from the language $\mathcal{L}_2$ is an SLTS, only containing transitions of the form $\xrightarrow{a,\lambda}$.

We return to our example of a queueing system. The arrival process is now modelled as follows, employing the Markovian action prefix:

$$Arrival := (arrive, \lambda);\ (enq, 1);\ Arrival$$

Action $arrive$ occurs with rate λ, whereas for action enq we specify rate 1, the neutral element of multiplication. Via synchronisation, the queue process determines the actual rate of enq.

$$\begin{aligned} Queue_0 &:= (enq, \eta);\ Queue_1 \\ Queue_i &:= (enq, \eta);\ Queue_{i+1}\ []\ (deq, \delta);\ Queue_{i-1} \qquad 1 \leq i < max \\ Queue_{max} &:= (deq, \delta);\ Queue_{max-1} \end{aligned}$$

Fig. 2 depicts the SLTS obtained from the parallel composition of processes $Arrival$ and $Queue_0$ synchronised over action enq.

From a given SLTS one can immediately construct a continuous time Markov chain (CTMC [16]). The arcs of the CTMC are given by the union of all the transitions joining the LTS nodes (regardless of their action labels), and the transition rate is the sum of the individual rates (this is

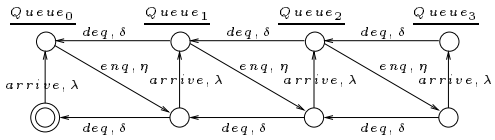

Figure 2: Semantic model of the Markovian queueing system, isomorphic to a CTMC

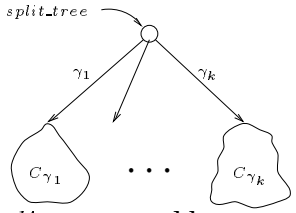

Figure 3: $split\_tree$ used by procedure $split'$

justified by the properties of the exponential distribution). Transitions leading back to the same node (loops) can be neglected, since they would have no effect on the balance equations of the CTMC. Performance measures can be derived by calculating the steady-state or transient state probabilities of the CTMC.

As already mentioned, both strong and weak Markovian bisimilarity coincide with Markovian bisimilarity à la MTIPP on this language. The technical reason is that the first clauses of Definition 2 and Definition 3 are irrelevant, while the respective second clauses both boil down to $\gamma(P, a, C) = \gamma(Q, a, C)$ for all actions a and classes C. This equivalence notion has a direct correspondence to the notion of *lumpability* on CTMCs [16, 14]. The basic bisimulation algorithm is the same as in Sec. 3, only the procedure $split$ needs to be modified. Procedure $split'$ now uses a data structure $split\_tree$ which is shown in Fig. 3. It essentially sorts states according to their γ-values. During refinement, when a class C is split by means of a splitter (a, C_{spl}), possibly more than two subclasses $C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}$ will be generated. Input class C is split such that the cumulative rate $\gamma(P, a, C_{spl}) = \gamma_j$ is the same for all the states P belonging to the same subclass C_{γ_j}, a leaf of the $split\_tree$.

procedure $split'(C, a, C_{spl}, Partition, Splitters)$
  forall $P \in C$
    $\gamma := \gamma(P, a, C_{spl})$ /* the cumulative rate from state P to C_{spl} is computed */
    $insert(split\_tree, P, \gamma)$ /* state P is inserted into the $split\_tree$ */
  /* now, $split\_tree$ contains k leaves $C_{\gamma_1}, \ldots, C_{\gamma_k}$ */
  if $(k > 1)$ /* only continue if C has been split into $k > 1$ subclasses */
    $Partition := Partition \cup \{C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}\} - \{C\}$
    $Splitters := Splitters \cup (Act \times \{C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}\}) - Act \times \{C\}$
    /* the partition and the splitter set are updated */

In the **forall** loop of procedure $split'$, the cumulative rate γ is computed for every state P in class C, and state P is inserted into the $split\_tree$ such that states with the same cumulative rate belong to the same leaf (procedure $insert$). The $split\_tree$ has k leaves, i.e. k different values of γ have appeared. If splitting has taken place (i.e. if $k > 1$), the partition must be refined and the set of splitters has to be updated.

The above algorithm computes Markovian bisimilarity on a given SLTS. It can be implemented such that the time complexity is of order $\mathcal{O}(m \log n)$ and the space complexity of order $\mathcal{O}(m + n)$, where n is the number of states and m the number of transitions. The proof is given in [8].

5 Symbolic representation with BDDs

In this section, we discuss details of a BDD-based implementation of the above algorithms. BDDs are graph-based representations of Boolean functions and have recently gained remarkable attention as efficient encodings of very large state spaces. In a process algebraic context, this efficiency

is mainly due to the fact that the parallel composition operator can be implemented on BDDs in such a way that the size of the data structure only grows linearly in the number of parallel components, especially for loosely coupled components. This compares favourably to the exponential growth caused by the usual operational semantics, due to the interleaving of causally independent transitions. We explain how LTSs can be encoded as BDDs and illustrate a way to include the rate information of SLTS into this data structure and the bisimulation algorithms. To complete the picture, we also discuss parallel composition on BDDs.

5.1 Binary Decision Diagrams and the encoding of LTSs

A Binary Decision Diagram (BDD) [2] is a symbolic representation of a Boolean function $f : \{0,1\}^n \to \{0,1\}$. Its graphical interpretation is a rooted directed acyclic graph, essentially a collapsed binary decision tree in which isomorphic subtrees are merged and "don't care" nodes are skipped (a node is called "don't care" if the truth value of the corresponding variable is irrelevant for the truth value of the overall function). It is known that BDDs provide a canonical representation for Boolean functions, assuming a fixed ordering of the Boolean variables. Algorithms for BDD construction from a Boolean expression and for performing Boolean operations (and, or, not, ...) on BDD arguments all follow a recursive scheme.

A LTS can be represented symbolically by a BDD. The idea is to encode states and actions by Boolean vectors (for the moment, we look at the non-stochastic case where it is not necessary to consider information about transition rates). One transition of the LTS then corresponds to a conjunction of $n_a + 2n_s$ literals (a literal is either a Boolean variable or the negation of a Boolean variable) $\bigwedge_{i=1}^{n_a} a_i \bigwedge_{j=1}^{n_s} s_j \bigwedge_{j=1}^{n_s} t_j$, where literals $a_1 \ldots a_{n_a}$ encode the action, $s_1 \ldots s_{n_s}$ identify the source state and $t_1 \ldots t_{n_s}$ the target state of the transition (we assume that the number of distinct actions to be encoded is between 2^{n_a-1} and $2^{n_a}+1$, so that n_a bits are suitable to encode them, and similarly for the number of states). The overall LTS corresponds to the disjunction of the terms for the individual transitions. The size of a BDD is highly dependent on the chosen variable ordering. In the context of transition systems, experience has shown that the following variable ordering yields small BDD sizes [5]:

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

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.

To illustrate the encoding, Fig. 4 shows the LTS corresponding to the $Queue_0$ process from Sec. 3 (assuming, again, that $max = 3$), the way transitions are encoded and the resulting BDD (in the graphical representation of a BDD, one-edges are drawn solid, zero-edges dashed, and for reasons of simplicity, the terminal false-node and its adjacent edges are omitted). Since there are only two different actions (enq and deq), one bit would be enough to encode the action. However, in view of action $arrive$ which will be needed for process $Arrival$, we use two bits to encode the action, i.e. $n_a = 2$. The LTS has four states, therefore two bits are needed to represent the state, i.e. $n_s = 2$. In the BDD, one can observe the interleaving of the Boolean variables for the source and target state.

The parallel composition operator can be realised directly on the BDD representation of the two operand processes. Consider the parallel composition of two processes, $P = P_1 \,|[A]|\, P_2$, and assume that the BDDs which correspond to processes P_1 and P_2 have already been generated and are denoted $\mathcal{P}_1$ and $\mathcal{P}_2$. The set A can also be coded as a BDD, namely $\mathcal{A}$. The BDD $\mathcal{P}$ which corresponds to the resulting process P can then be written as a Boolean expression:

$$\begin{aligned} \mathcal{P} \;=\; & (\mathcal{P}_1 \wedge \mathcal{A}) \wedge (\mathcal{P}_2 \wedge \mathcal{A}) \\ & \vee\; (\mathcal{P}_1 \wedge \overline{\mathcal{A}} \wedge Stab_{P_2}) \;\vee\; (\mathcal{P}_2 \wedge \overline{\mathcal{A}} \wedge Stab_{P_1}) \end{aligned}$$

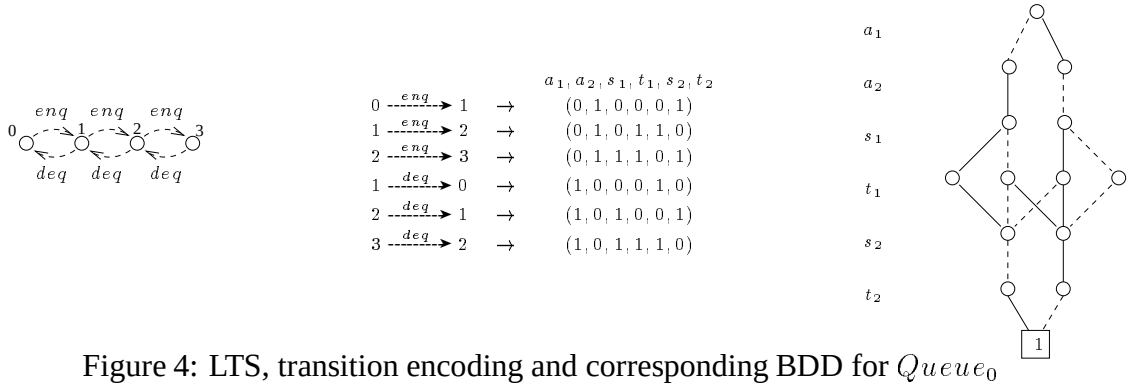

Figure 4: LTS, transition encoding and corresponding BDD for $Queue_0$

The term on the first line is for the synchronising actions in which both P_1 and P_2 participate. The first (second) term on the second line is for those actions which P_1 (P_2) performs independently of P_2 (P_1) — these actions are all from the complement of A. The meaning of $Stab_{P_2}$ $(Stab_{P_1})$ is a BDD which expresses stability of the non-moving partner of the parallel composition, i.e. the fact that the source state of process P_2 (P_1) equals its target state.

The BDD resulting from the parallel composition, $\mathcal{P}$, describes all transitions which are possible in the product space of the two partner processes. Given a pair of initial states for P_1 and P_2, only part of the product space may be reachable due to synchronisation constraints. Reachability analysis can be performed on the BDD representation, restricting $\mathcal{P}$ to those transitions which originate in reachable states.

5.2 Symbolic bisimulation

The basic bisimulation algorithm of Sec. 3 and its various optimisations can be realised efficiently using BDD-based data structures. For convenience, the transition system is represented not by a single BDD, but by a set of BDDs $T_a(s,t)$, one for each action a (here, s and t denote vectors of Boolean variables of length n_s). The current partition is stored as a set of BDDs $\{C_1(s), C_2(s), \ldots\}$, one for each class. When class C is split into subclasses C^+ and C^- during execution of procedure $split$, those subclasses are also represented by BDDs. The dynamic set of splitters, $Splitters$, is realised as a pointer structure. The computation of the subclass C^+ in procedure $split$ is formulated as a Boolean expression on BDD arguments

$$C^+(s) := C(s) \wedge \exists\, t :\ (T_a(s,t) \wedge C_{spl}(t))$$

where the existential quantification is also performed on BDDs.

5.3 BDDs with rate information

Clearly, pure BDDs are not capable of representing the numerical information about the transition rates of a *stochastic* LTS. In the literature, several extensions of the BDD data structure have been proposed for representing functions of the type $f : \{0,1\}^n \rightarrow I\!R$. Most prominent among these are multi-terminal BDDs [4], edge-valued BDDs [17] and Binary Moment Diagrams (BMD) [3]. In all of these approaches, the basic BDD structure is modified and the efficiency of the data structure, due to the sharing of isomorphic subtrees, may be diminished. Based on this observation, we developed a different approach which we call decision-node BDD (DNBDD) [20]. The distinguishing feature of DNBDDs is that the basic BDD structure remains completely untouched when moving from an LTS encoding to an SLTS encoding. The additional rate information is attached to specific edges of this BDD in an orthogonal fashion.

In a BDD representing a LTS, a *path* p from the root to the terminal true-node corresponds to 2^k transitions of the transition system, where k is the number of "don't care" variables on that path. Since these transitions are labelled by 2^k distinct rates, we need to assign a rate list of length 2^k to that path. Let $rates(p)$ denote a list of real values $(\lambda_0, \ldots, \lambda_{2^k-1})$, where k is the number of "don't cares" on path p. The correspondence between transitions and individual rates of such a

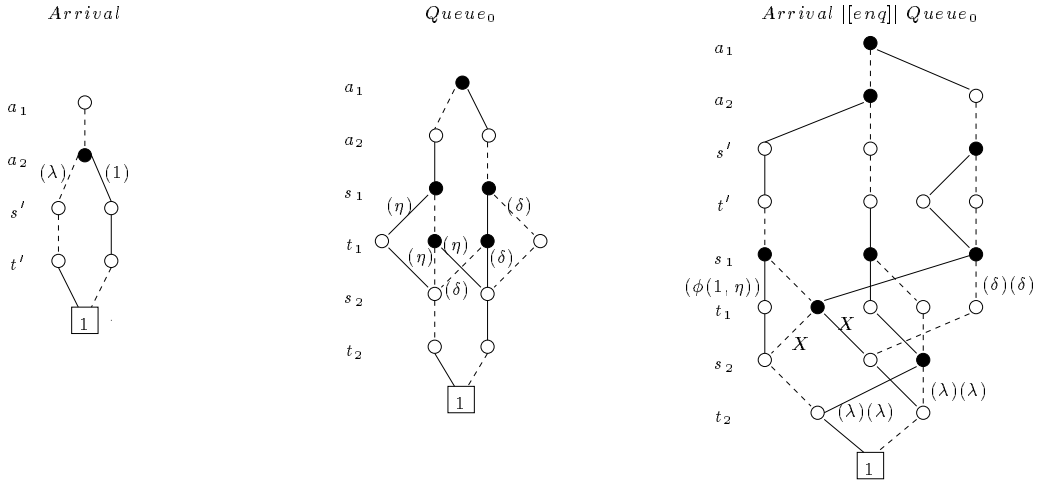

Figure 5: DNBDDs for the queueing example (shorthand notation: $X = (\phi(1,\eta))(\delta)(\delta)$)

list is implicitly given by the valuation of the encoding of the transitions on "don't care" nodes, which ranges from 0 to $2^k - 1$. For the practical realisation of this concept, and in order to make our representation canonical, we must answer the question of where to store the rate lists. This leads to the following consideration: Instead of characterising a path by all its nodes, we observe that a path is fully characterised by its *decision nodes*.

Definition 4 *A* decision node *is a non-terminal BDD node whose successor nodes are both different from the terminal false-node. A* decision node BDD (DNBDD) *is a BDD enhanced by a function*

$$rates : Paths \rightarrow (\mathbb{R})^+$$

where $Paths$ is the set of paths from the root node to the terminal true-node (and $(\mathbb{R})^+$ is the set of finite lists of real values), such that for any such path p,

$$rates(p) \in (\mathbb{R})^{2^k}$$

if k is the number of "don't cares" on path p. The list $rates(p) = (\lambda_0, \ldots, \lambda_{2^k-1})$ is attached to the outgoing edge of the last decision node on path p, i.e. the decision node nearest to the terminal true-node.

To illustrate the DNBDD concept, we return to our queueing example. Fig. 5 shows the DNBDDs associated with processes $Arrival$, $Queue_0$ and $Arrival \ ||[enq]|| \ Queue_0$ (in the figure, decision nodes are drawn black). On the left, rates λ and 1 are attached to the outgoing edges of the (single) decision node of the BDD. In the middle, six individual rates are attached to the appropriate edges. On the right, up to three rate lists, each consisting of a single rate, are attached to BDD edges. For instance, the rate lists $(\delta)(\delta)$ specify the rates of the two transitions encoded as bitstrings 10110010 and 10000010 whose paths share the last decision node.

In the case where several rate lists are attached to the same BDD edge (because several paths share their last decision node) it is important to preserve the one-to-one mapping between paths and rate lists. This could simply be accomplished by the lexicographical ordering of paths. For algorithmic reasons, however, we use a so-called rate tree, an unbalanced binary tree which makes it possible to access rate lists during recursive descent through the BDD [20]. In our current implementation of DNBDDs, the rate tree is implemented as illustrated in Fig. 6. This figure (left) shows the encoding of the transitions of some SLTS, each of the transition being associated with a rate. The first two transitions share the same path, a path which has a "don't care" in the Boolean variable s. Therefore, the corresponding rate list (λ_0, λ_1) has length two. The other four paths do not have any "don't care" variables, they each correspond to exactly one transition of the SLTS and the corresponding rate lists have length one. The latter four paths all share their last decision node. Therefore each of the outgoing edges of that decision node carries two rate lists (of length one). The rate tree is built as a separate data structure from the BDD. However, its internal nodes

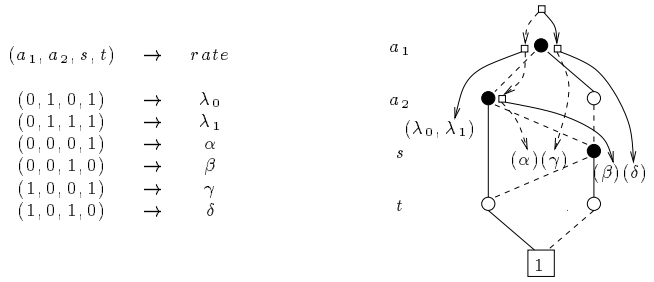

Figure 6: Encoded transitions with rates, and corresponding DNBDD with rate tree

and the rate lists are associated with the decision nodes of the BDD as indicated in Fig. 6 (right). The rate tree is manipulated by an appropriate extension of the procedures which manipulate the BDD. This implementation of the rate tree has the drawback that it requires the explicit storage of one rate for each encoded transition which may cause considerable overhead. In order to avoid such redundancies, an efficient data structure to represent rate trees might itself be based on BDDs. We are currently investigating this issue.

Parallel composition of two SLTSs based on their symbolic representation follows the same basic algorithm as sketched in Sec. 5.1. Similar to the fact that the operational rules in Sec. 2 are parametric in the synchronisation policy, the concept of DNBDDs is not bound to a particular choice of function ϕ, any arithmetic expression of the two partner rates can be employed.

5.4 Symbolic Markovian bisimulation

We now discuss aspects of a DNBDD-based algorithm which computes Markovian bisimulation on SLTSs. The basic algorithm is the same as in Sec. 4, only procedure $split'$ needs to be adapted. When using DNBDDs, the cumulative rate of action a from state P to class C_{spl} is computed as follows: We compute $T_{P \stackrel{a}{\rightarrow} C_{spl}}(s,t)$, the DNBDD which represents all a-transitions from state P to states from class C_{spl}. It can be obtained by restricting $T_a(s,t)$ to the single source state P and to target states from class C_{spl} (again, the transition relation is represented by individual DNBDDs $T_a(s,t)$, one for every action a, and class C is represented by a BDD $C(t)$). The cumulative rate $\gamma(P, a, C_{spl})$ is computed by applying the function $sum\_of\_all\_rates$ to $T_{P \stackrel{a}{\rightarrow} C_{spl}}(s,t)$. This function simply sums up all the entries of all rate lists of a DNBDD (e.g., application of this function to the DNBDD in Fig. 6 yields $\lambda_0+\lambda_1+\alpha+\beta+\gamma+\delta$). Furthermore, in the $split\_tree$ used by procedure $split'$ the subclasses $C_{\gamma_1}, \ldots, C_{\gamma_k}$ are now also represented by BDDs.

procedure $split'(C, a, C_{spl}, Partition, Splitters)$
  forall $P \in C$
    $T_{P \stackrel{a}{\rightarrow} C_{spl}}(s,t) := T_a(s,t) \wedge (s \ddot{=} P) \wedge C_{spl}(t)$ /* $s \ddot{=} P$ denotes that state P is encoded as s */
    $\gamma := sum\_of\_all\_rates(T_{P \stackrel{a}{\rightarrow} C_{spl}}(s,t))$ /* the cumulative rate from P to C_{spl} is computed */
    $insert(split\_tree, P, \gamma)$ /* state P is inserted into the $split\_tree$ */
  /* now, $split\_tree$ contains k leaves $C_{\gamma_1}, \ldots, C_{\gamma_k}$ */
  if $(k > 1)$
    ... /* the remaining part of procedure $split'$ is as in Sec. 4, */
    /* but $Partition$ and $Splitters$ are represented as BDDs */

6 Conclusion

In this paper, we have discussed efficient algorithms to compute bisimulation style equivalences for Stochastic Process Algebras. In addition, we have presented details of a BDD-based implementation of these algorithms, describing DNBDDs to represent the additional rate information which is relevant for the analysis of the underlying Markov chain.

The usefulness of BDDs to encode transition systems has been stressed by many authors. However, we would like to point out that the myth, saying that BDDs always provide a more compact encoding than the ordinary representation (as a list or a sparse matrix data structure), does not hold

in general. A naïve encoding of transition systems as BDDs does not save space. Heuristics for encodings are needed, exploiting the structure of the specification. The implementation of parallel composition on BDDs is indeed such a heuristics, and a very successful one, since an exponential blow-up can be turned into a linear growth.

Apart from encoding transition systems as (DN)BDDs and parallel composition on (DN)BDDs, we have described how bisimulation algorithms can be implemented on these data structures. As a consequence, all the ingredients are at hand for carrying out compositional aggregation of SPA specifications in a completely BDD-based framework. In this way, the state space explosion problem can be alleviated. We are currently implementing all these ingredients in a prototypical tool written in C, based on our own DNBDD package. However, in order to obtain performance results, the (minimised) BDD representation still has to be converted back to the ordinary representation, since we do not yet have a Markov chain analyser which works directly on DNBDDs. Numerical analysis based on DNBDDs is one of our topics for future work. For this purpose, it seems beneficial to investigate the relation between DNBDDs and MTBDDs, since MTBDD-based numerical analysis methods have already been developed [7, 13].

References

[1] T. Bolognesi, E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems* 14:25-59, 1987.

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

[3] R.E. Bryant, Y. Chen. Verification of Arithmetic Functions with Binary Moment Diagrams. In *Proc. 32nd Design Automation Conference*, 535-541, ACM/IEEE, 1995.

[4] E.M. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, X. Zhao. Multi-terminal Binary Decision Diagrams: An efficient data structure for matrix representation. In *Proc. Int. Workshop on Logic Synthesis*, Tahoe City, May 1993.

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

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

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

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

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

[10] H. Hermanns, M. Rettelbach. Syntax, Semantics, Equivalences, and Axioms for MTIPP. In *Proc. 2nd PAPM Workshop*. University of Erlangen-Nürnberg, IMMD 27(4):71-87, 1994.

[11] H. Hermanns, J.P. Katoen. Automated Compositional Markov Chain Generation for a Plain Old Telephony System. to appear in *Science of Computer Programming*, 1998.

[12] H. Hermanns, M. Siegle. Bisimulation Algorithms for Stochastic Process Algebras and their BDD-based Implementation. In *Proc. 5th Int. AMAST Workshop on Real-Time and Probabilistic Systems*, LNCS, 1999.

[13] H. Hermanns, J. Meyer-Kayser, M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. submitted for publication, 1999.

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

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

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

[17] Y.-T. Lai, S. Sastry. Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification. In *29th Design Automation Conference*,608-613, ACM/IEEE, 1992.

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

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

[20] M. Siegle. Technique and tool for symbolic representation and manipulation of stochastic transition systems. TR IMMD 7 2/98, Universität Erlangen-Nürnberg, March 1998.