# Bisimulation Algorithms for Stochastic Process Algebras and their BDD-based Implementation[⋆]

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, IMMD, 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, various proposals for compositional aggregation have been made. They rely on minimisation with respect to a congruence relation. 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

Compositional application of stochastic process algebras (SPA) 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. Examples of such relations are strong equivalence [22], (strong and weak) Markovian bisimilarity [16] and extended Markovian bisimilarity [2]. 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, for instance, a telephony system model [18]. Without compositional aggregation, the state space turned out to consist of more than 10 million states, while only 720 states were actually required using compositional aggregation.

Applicability of compositional aggregation relies on the existence of *algorithms* to compute minimised components. In this paper, we discuss efficient algorithms for strong equivalence, and (strong and weak) Markovian bisimulation. The algorithms are variants of well-known partition refinement algorithms [29, 11, 24]. They compute partitions of equivalent states of a given state space by iterative refinement of partitions, until a fixed point is reached.

For the practical realisation of the algorithms we introduce BDD-based data structures. During the recent years, BDDs [6] have been shown to enable an efficient, *symbolic* encoding of state spaces. In particular, the parallel composition operator can be defined on BDDs in a way which avoids the usually observed exponential blow-up due to interleaving of causally independent transitions [10]. In this paper, we highlight how parallel composition and compositional aggregation can both be performed symbolically in a stochastic setting.

---

[⋆] A preliminary version of this paper has been presented at the PAPM'98 workshop [20].

This paper is organised as follows: Sec. 2 contains the definition of the languages and of the bisimulation relations which we consider. Sec. 3 presents the basic bisimulation algorithm for non-stochastic process algebras. Sec. 4 and Sec. 5 do the same for the purely Markovian case and for the case where both Markovian and immediate transitions are allowed. In Sec. 6, we focus on BDDs and introduce a novel stochastic extension, DNBDDs. Furthermore, we show how algorithms for parallel composition and bisimulation can benefit from the use of these data structures. The paper concludes with Sec. 7.

## 2 Basic definitions

This section introduces the scenario that will be considered in the sequel. We define a language and its operational semantics. In addition, we recall the notions of strong and weak Markovian bisimilarity. We refer to [16] for more details and motivating examples.

**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:*

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

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

The following operational semantic rules define a labelled transition system (LTS) containing action transitions, $\dashrightarrow^{a}\dashrightarrow$, and Markovian transitions, $\xrightarrow{a,\lambda}$. The semantic rule for synchronisation of Markovian transitions is parametric in a function $\phi$ determining the rate of synchronisation, in response to the fact that different synchronisation policies (minimum, maximum, product, . . . ) are possible. Note, however, that the apparent rate construction of PEPA [22] requires a function $\phi(P, Q, \lambda, \mu)$ instead.

Strong and weak Markovian bisimilarity are defined in a variant of Larsen & Skou style [27], using the function $\gamma : \mathcal{L} \times Act \times 2^{\mathcal{L}} \mapsto I\!\!R$, often called the *cumulative rate*, defined as follows (we use $\{\!|$ and $|\!\}$ to denote multiset brackets):

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

**Definition 2.** *An equivalence relation $\mathcal{B}$ is a strong Markovian bisimulation, if $(P, Q) \in \mathcal{B}$ implies that*
*(i) $P \dashrightarrow^{a} P'$ implies $Q \dashrightarrow^{a} Q'$, 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 $\dashrightarrow^{a}$ with $\Longrightarrow^{a}$. Here, $\Longrightarrow^{a}$ denotes an observable $a$ transition that is preceded and followed by an arbitrary number (including zero) of invisible activities, i.e. $\Longrightarrow^{a} :=$ $\dashrightarrow^{i^*} \dashrightarrow^{a} \dashrightarrow^{i^*}$. If $a$ is internal ($a = \mathtt{i}$), $\Longrightarrow^{a}$ abbreviates $\dashrightarrow^{i^*}$. As discussed in [16], 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 [15].

**Definition 3.** *An equivalence relation $\mathcal{B}$ is a weak Markovian bisimulation, if $(P, Q) \in \mathcal{B}$ implies that*
*(i) $P \Longrightarrow^{a} P'$ implies $Q \Longrightarrow^{a} Q'$, for some $Q'$ with $(P', Q') \in \mathcal{B}$,*
*(ii) if $P \Longrightarrow^{i} P' \not\dashrightarrow^{i}$ then there exists $Q'$ such that $Q \Longrightarrow^{i} Q' \not\dashrightarrow^{i}$, 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.*

In this definition, $P \not\dashrightarrow^{i}$ 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 $P \dashrightarrow^{i}$).

It can be shown that strong Markovian bisimilarity is a congruence with respect to the language operators, provided that $\phi$ is distributive over summation of real values. The same result holds for weak Markovian bisimilarity except for congruence with respect to choice, see [15].

In the sequel, we consider two distinct sub-languages of $\mathcal{L}$. The first, $\mathcal{L}_1$, arises by disallowing Markovian prefix. This sub-language gives rise to an ordinary, non-stochastic process algebra, a subset of Basic LOTOS [3] 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 [28]. The complementary subset, $\mathcal{L}_2$, is obtained by disallowing the other prefix, action prefix. The resulting language coincides with MTIPP à la [17] (if $\phi$ 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
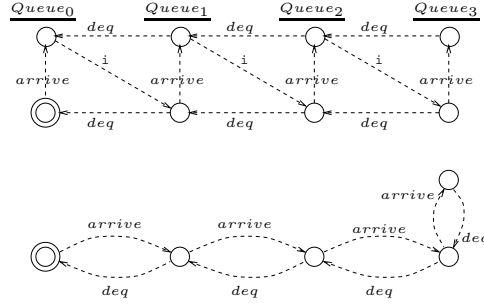
**Fig. 1.** LTS of the queueing system example, before and after applying weak bisimilarity

equivalence [22]. The semantics of $\mathcal{L}_2$ only contains Markovian transitions, and we will refer to such a transition system as a stochastic LTS (SLTS). The complete language, where both prefixes coexist involves both types of transitions, and we shall call such a transition system an extended SLTS (ESLTS).

## 3 Bisimulation minimisation in non-stochastic process algebras

In this section, we introduce the general idea of iterative partition refinement, working with the language $\mathcal{L}_1$. We aim to set the ground for an understanding of the following sections. 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.

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

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.

$$System := \textbf{hide} \; enq \; \textbf{in} \; \Big( Arrival \; |[enq]| \; Queue_0 \Big)$$

Fig. 1 (top) 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 (bottom) 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. Traditionally, they follow an iterative refinement scheme [29, 11, 24]. 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. The algorithm 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 [11, 29]. 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.

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** $C \in Partition$      $split(C, a, C_{spl}, Partition, Splitters)$
      /* 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** $split(C, a, C_{spl}, Partition, Splitters)$
   $C^+ := \{P \mid P \in C \ \wedge \ \exists Q : (P \dashrightarrow^{a} Q \ \wedge \ Q \in C_{spl})\}$
   /* 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 */

We illustrate the algorithm by means of the above queueing example. In fact, we shall compute *weak* instead of strong bisimilarity. The only change that is necessarry for this purpose concerns the transition relation $\dashrightarrow^{a}$ used in procedure $split$, which is replaced by the weak relation $\Longrightarrow^{a}$. However, this requires the computation of $\Longrightarrow^{a}$ during the initialisation phase. As a matter of fact, the computation of $\Longrightarrow^{a}$ dominates the complexity of partition refinement, basically because the reflexive and transitive closure $\dashrightarrow^{i^*}$ of internal moves has to be computed in order to build the weak transition relation. The usual way of computing a transitive closure has cubic complexity. (Some slight improvements are known for this task, see for instance [9]. In any case, this is the computationally expensive part.)
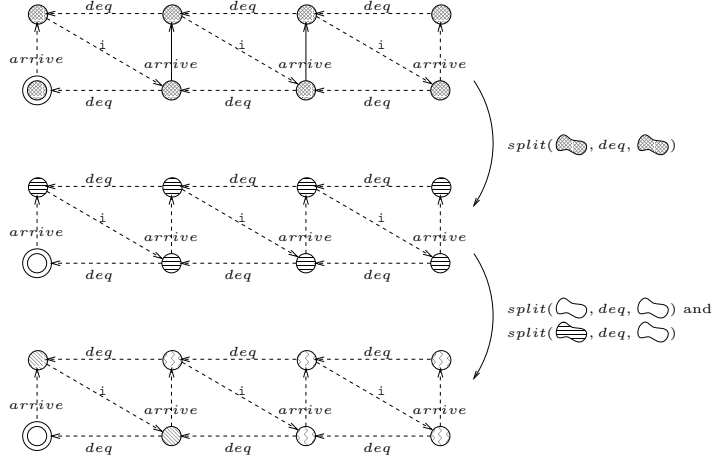
**Fig. 2.** Initialisation, first and second refinement step of the algorithm

The LTS is depicted in Fig. 2 (top) where we have used a particular shading of states in order to visualize the algorithm. In the beginning all states are assumed to be equivalent, and hence, all states are shaded with the same pattern. We use ⬬ to refer to the set of states shaded like ◉. So, $Partition := \{⬬\}$, and $Splitters$ is initialised accordingly.

After computing the weak transition relation $\dashrightarrow$, we start partition refinement by choosing a splitter, say $(deq, ⬬)$ and computing $split(⬬, deq, ⬬)$. The initial state has no possibility to perform a $\overset{deq}{\dashrightarrow}$ transition in contrast to all other states. Therefore $⬬^{+} = ⬭$ and $⬬^{-} = ⬯$. As a consequence, $Partition$ becomes $\{⬯, ⬭\}$ and new splitters are added to $Splitters$ while the currently processed one, $(deq, ⬬)$, is removed. This completes the first iteration and leads to the situation depicted in Fig. 2 (middle).

By choosing a different splitter, say $(deq, ⬯)$, we start the next iteration. Since $Partition$ now contains two elements, we compute $split(⬯, deq, ⬯)$ and $split(⬭, deq, ⬯)$. $⬯$ cannot be split any further, while splitting of $⬭$ returns $⬭^{+} = ⬬$ and $⬭^{-} = ⬰$. Updating $Partition$ to $\{⬯, ⬬, ⬰\}$ and adding new splitters leads to the situation depicted in Fig. 2 (bottom). Subsequent iterations of the algorithm will divide ⬰ further, leading to five partitions in total. The algorithm terminates once the set $Splitters$ is empty.

## 4 The Markovian case

In this section, we consider the MTIPP-style language $\mathcal{L}_2$ where *all* actions are associated with a delay which is an exponentially distributed random variable. In addition, MTIPP instantiates $\phi$ with the product of rates, for reasons discussed (for instance) in [16]. The semantic model of a process from the language $\mathcal{L}_2$ is an SLTS, only containing transitions of the form $\xrightarrow{a,\lambda}$.
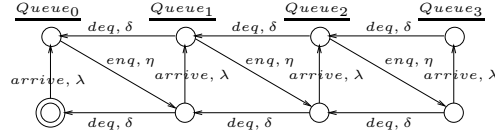
**Fig. 3.** Semantic model of the Markovian queueing system, isomorphic to a CTMC

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 $\lambda$, whereas for action $enq$ we specified the (passive) rate 1, the neutral element of multiplication. The queue process determines the actual rate of $enq$, occuring as a result of synchronisation via $enq$.

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

Fig. 3 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 [25]). The arcs of the CTMC are given by the union of all the transitions joining the LTS nodes (regardless of their labels), and the transition rate is the sum of the individual rates. This is justified by the properties of the exponential distribution, in particular the fact that the minimum of two exponentially distributed random variables with rates $\lambda_1, \lambda_2$ is again exponentially distributed with rate $\lambda_1 + \lambda_2$. Transitions leading back to the same node (loops) can be neglected, since they would have no effect on the balance equations of the CTMC. The CTMC carries only the (cumulated) rate labels. Performance measures can then 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 [25, 22]. As a consequence, the algorithm which we develop can be used to efficiently compute *lumpable partitions* of an SPA description (as well as a CTMC in isolation). 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. 4. It essentially sorts states according to their $\gamma$-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 $C_{\gamma_j}$, a leaf of the $split\_tree$.

**procedure** $split'(C, a, C_{spl}, Partition, Splitters)$
    **forall** $P \in C$
        $\gamma := \gamma(P, a, C_{spl})$
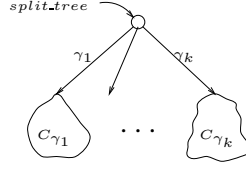        /* the cumulative rate from state $P$ to $C_{spl}$ is computed */

**Fig. 4.** *split_tree* used by procedure *split'*

$insert(split\_tree, P, \gamma)$
/* 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 $\gamma$ 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 $\gamma$ 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.

**Theorem 1.** *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 is of order $\mathcal{O}(m + n)$, where $n$ is the number of states and $m$ is the number of transitions.*

The detailed proof is given in [15]. As in [23], the crucial point to obtain the time complexity of non-stochastic strong bisimilarity is that all but one of the largest subclasses $C_{\gamma_i}$ are actually inserted into the set $Splitters$ of potential splitters. (No assumptions about the time complexity of arithmetic operations are required).

## 5 Markovian and immediate actions

In this section, we consider the complete language $\mathcal{L}$ where both immediate and Markovian actions coexist. Again, we return to our queueing system example. In the arrival process, action $arrive$ has an exponential delay, whereas action $enq$ is immediate.
$$Arrival := (arrive, \lambda); \ enq; \ Arrival$$
The specification of the Queue is again modified with respect to Sec. 3, i.e. action $enq$ is immediate and action $deq$ has exponential delay.
$$Queue_0 := enq; \ Queue_1$$
$$Queue_i := enq; \ Queue_{i+1} \ [] \ (deq, \delta); \ Queue_{i-1} \qquad 1 \le i < max$$
$$Queue_{max} := (deq, \delta); \ Queue_{max-1}$$
The overall system is again given by the composition of $Arrival$ and $Queue_0$, where $enq$ is hidden after synchronisation. The semantic model of such a specification from the language $\mathcal{L}$ is an ESLTS with two types of transitions: Markovian transitions $\longrightarrow$,
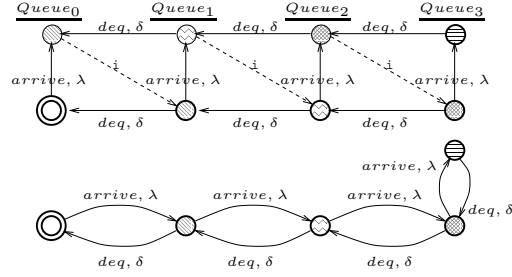
**Fig. 5.** ESLTS of the queueing system example, before and after applying weak Markovian bisimilarity

and action transitions - - -➤. Fig. 5 (top) depicts the ESLTS for the example queueing system. In the context of our complete language $\mathcal{L}$, the notion of weak Markovian bisimilarity is central for associating a CTMC to a given specification. The reason is that immediate transitions do not have a counterpart on the level of the CTMC. Weak Markovian bisimilarity justifies to eliminate *internal* immediate transitions such that a SLTS, and hence an (action labelled) CTMC results from the quotient transition system (the transition system obtained by representing each equivalence class by a single state). The details of this strategy are described in [18, 15]. In order to illustrate how the relation can be used to achieve this, the equivalence classes of weak Markovian bisimilarity are indicated in Fig. 5 (bottom). Note, however, that this effect relies on abstraction of immediate actions before applying weak Markovian bisimilarity. Furthermore, a unique CTMC exists only if nondeterminism is absent after applying weak Markovian bisimilarity. The absence of nondetermininsm is easy to check on the quotient transition system.

Accordingly, an algorithm to compute weak Markovian bisimilarity is central for deriving the quotient transition system, and hence the CTMC. (In principle, the CTMC could also be derived from a specification by purely syntactic transformations, using the complete axiomatisation from [19].) Our algorithm is based on the one given in the previous section, but proceeds in a different way. The technical reason is that, similar to the computation of branching bisimilarity [12], refining a partition by means of a splitter might cause that the refinement with respect to already processed splitters has to be repeated for this partition.[1] This is a crucial difference with respect to the algorithms we have described before, for which termination is only guaranteed because refinement with respect to an already processed splitter is never repeated for that splitter. For branching bisimilarity, the problem is tackled in [13], by introducing a distinct treatment of vanishing states. Bouali has adopted this machinery to compute also weak bisimilarity [4]. Indeed our algorithm is based on this adaption.

We use $P \searrow^i P'$ to indicate that $P$ may internally and immediately evolve to a tangible state $P'$, i.e. where no further internal immediate transition is possible. Formally, $P \searrow^i P'$ iff $P \Longrightarrow^i P'$ for some $P'$ with $P' \not\xrightarrow{i}$. If there is at least one tangible

---

[1] In terms of [13], stability is not inherited under refinement.

state $P'$ that can be reached from $P$ via a (possibly empty) sequence of internal immediate transitions we use the predicate $P \searrow^{\mathbf{i}}$. Note that $P \searrow^{\mathbf{i}}$ whenever $P$ is tangible. The converse situation, where a (vanishing) state $P$ has no possibility to internally and immediately evolve to a tangible state, is denoted $P \not\searrow^{\mathbf{i}}$. We refer to such states as *divergent* states, as opposed to *convergent* states (satisfying $P \searrow^{\mathbf{i}}$). For the presentation of the algorithm, we first restrict to transition systems that do not contain divergent states. This is done to simplify the exposition, the general case is discussed afterwards. Restricted to divergence-free ESLTS, the basic algorithm is as follows.

1. **Initialisation** as before in Sec. 3. In addition, weak transitions ===➤ are computed from ---➤.
2. **Main loop**
   **while** ($Splitters \neq \emptyset$)
     **choose** splitter $(a, C_{spl})$
     **forall** $C \in Partition$     $split(C, a, C_{spl}, Partition, Splitters)$
     /* all classes are split with respect to weak transitions */
     **forall** $C \in Partition$     $split''(C, a, C_{spl}, Partition, Splitters)$
     /* all classes are split with respect to Markovian transitions */
     $Splitters := Splitters - (a, C_{spl})$
     /* the processed splitter is removed from the splitter set */

The main loop contains two different procedures, $split$ and $split''$ requiring further explanation. The first, $split(C, a, C_{spl}, Partition, Splitters)$, refines with respect to clause *(i)* of Definition 3. This is achieved using the procedure $split$ of Section 3, but applied on weak transitions, as in the example of Sec. 3. The second procedure, $split''(C, a, C_{spl}, Partition, Splitters)$, is more complicated. It refines with respect to the second clause of Definition 3. The details are given below.

**procedure** $split''(C, a, C_{spl}, Partition, Splitters)$

  **forall** $P \in C \ \wedge \ P \not\xrightarrow{\mathbf{i}}$
    /* $P$ is a tangible state */
    $\gamma := \gamma(P, a, C_{spl})$
    /* the cumulative rate to $C_{spl}$ is computed */
    $insert(split\_tree, P, \gamma)$
    /* state $P$ is inserted into the $split\_tree$ */
  /* now, $split\_tree$ contains $k$ leaves $C_{\gamma_1}, \ldots, C_{\gamma_k}$ */

  **forall** $P \in C \ \wedge \ P \xrightarrow{\mathbf{i}}$
    /* $P$ is a vanishing state */
    **if** $(\exists \, \gamma_j : \ P \searrow^{\mathbf{i}} Q \ \Rightarrow \ Q \in C_{\gamma_j})$
      /* $P$ can internally and immediately reach tangible states of class $C_{\gamma_j}$ only */
      $insert(split\_tree, P, \gamma_j)$
  $Partition := Partition \cup \{C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}\} - \{C\}$
  $Splitters := Splitters \cup (Act \times \{C_{\gamma_1}, C_{\gamma_2}, \ldots, C_{\gamma_k}\}) - Act \times \{C\}$
  /* the partition and the splitter set are updated */
  **if** $(C \neq \bigcup_1^k C_{\gamma_j})$
    /* some vanishing states have not been covered yet */
    $Partition := Partition \cup \{C - \bigcup_1^k C_{\gamma_j}\}$
    $Splitters := Splitters \cup (Act \times \{C - \bigcup_1^k C_{\gamma_j}\})$
    /* all remaining vanishing states form a new class, since they can internally
      and immediately evolve to tangible states belonging to different classes */
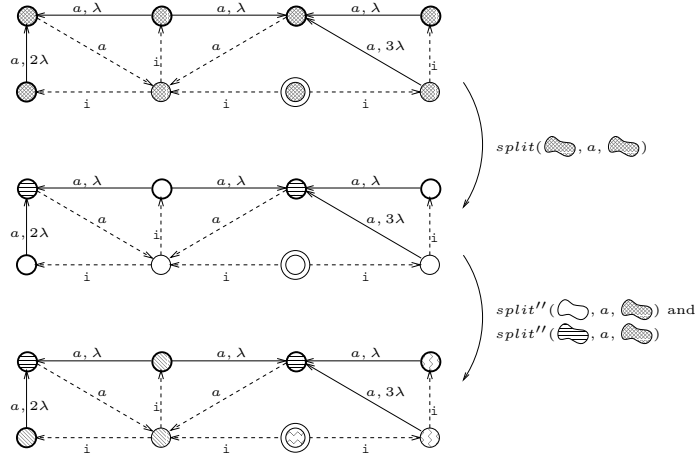
**Fig. 6.** Initialisation and first refinement step of the algorithm

The reader is invited to check the result depicted in Fig. 5 by means of this algorithm. In order to facilitate the inspection, tangible states are highlighted by bold circles in the figure. In order to illustrate the algorithm on a nontrivial example, where all the distinctions between different types of vanishing states matter, we consider a different example, depicted in Fig. 6 (top).

The initialisation proceeds as usual, after computing the weak transition relation ═══▸, we start the algorithm by choosing a splitter, say $(a, \text{⬗})$ and computing $split(\text{⬗}, a, \text{⬗})$. Two states may perform a ══$\overset{a}{}$══▸ transition in contrast to all other states. Therefore $\text{⬗}^+ = \text{⬗}$ and $\text{⬗}^- = \text{⬗}$. *Partition* and *Splitters* are updated accordingly, leading to the situation depicted in Fig. 6 (middle).

The subsequent invocation of $split''(\text{⬗}, a, \text{⬗})$ is most interesting (as opposed to $split''(\text{⬗}, a, \text{⬗})$). First, the three tangible markings (indicated by bold circles) in class $\text{⬗}$ are inserted into $split\_tree$, according to their cumulative rates of moving into (former) class $\text{⬗}$. This leads to a tree with two leaves, $C_{2\lambda}$ and $C_\lambda$, containing two, respectively one state. Now the three remaining vanishing states are treated: The righmost vanishing states can internally and immediately evolve only to tangible states of class $C_\lambda$ (note that according to Definition 3 the transition $\xrightarrow{a,3\lambda}$ is irrelevant since it originates in a vanishing state). For the same reason, the left vanishing state is inserted into class $C_{2\lambda}$. Only the initial state is not covered yet, since it has an internal, nondeterministic choice of behaving as a member of either of the classes. Hence, this state forms a new class, $\text{⬗}$ . In total, $split''(\text{⬗}, a, \text{⬗})$ has split $\text{⬗}$ into $\text{⬗}$, $\text{⬗}$ (representing $C_{2\lambda}$ and $C_\lambda$), and $\text{⬗}$ , leading to the situation depicted in Fig. 6 (bottom)(the *Partition* and *Splitters* are updated accordingly). This situation incidentally coincides with the classes of weak Markovian bisimilarity, because subsequent refinement steps do not reveal any distinction in one of these four classes. The algorithm terminates once the set *Splitters* is emptied.

To overcome the restriction to divergence-free ESLTS, a few modifications in the initialisation and the main loop of the algorithm are necessary. Algorithmically, the second clause of Definition 3 needs not to be checked at all for divergent states, while the first clause is still relevant. Furthermore, the second clause of Definition 3 implies that no convergent state is weakly Markovian bisimilar to a divergent state. These facts justify to (1) separate convergent and divergent states during inititalisation, and to (2) exclude the refinement of classes of divergent states by means of procedure $split''$ in the main loop of the algorithm. Recall that $split''$ implements refinement with respect to the second clause of Definition 3. So, for the general case, the algorithm becomes as follows, where $split$ and $split''$ are as before:

1. **Initialisation**
   Weak transitions $\Longrightarrow$ are computed from $\dashrightarrow$.
   $Con := \{\{P \in S \mid P \overset{i}{\searrow}\}\}$
   $Div := \{\{P \in S \mid P \overset{i}{\nsearrow}\}\}$
   /* the initial partition consists of two disjoint classes */
   $Splitters := Act \times (Con \cup Div)$
   /* 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** $C \in Con$     $split(C, a, C_{spl}, Con, Splitters)$
       /* all classes of convergent states are split with respect to weak transitions */
       **forall** $C \in Div$     $split(C, a, C_{spl}, Div, Splitters)$
       /* all classes of divergent states are split with respect to weak transitions */
       **forall** $C \in Con$     $split''(C, a, C_{spl}, Con, Splitters)$
       /* only the classes of convergent states are split with respect to Markovian transitions */
       $Splitters := Splitters - (a, C_{spl})$
       /* the processed splitter is removed from the splitter set */

An implementation of this algorithm, based on [23, 13, 4, 1] has a cubic time complexity.

**Theorem 2.** *The above algorithm computes weak Markovian bisimilarity on a given ESLTS. It can be implemented such that it requires $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space, where $n$ is the number of states.*

The proof is given in [15], for the divergence-free as well as the general case. It is worth pointing out that non-stochastic weak bisimulation essentially has the same complexity, due to the fact that a transitive closure operation is needed to compute weak transitions in either case.

## 6 Symbolic representation with BDDs

In this section, we discuss details of a BDD-based implementation of the above algorithms. BDDs are specific 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 (E)SLTS into this data structure and the bisimulation algorithms. To complete the picture, we also discuss parallel composition on BDDs.

### 6.1 Binary Decision Diagrams and the encoding of LTSs

A Binary Decision Diagram (BDD) [6] 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 [10]:
$$a_1 < \ldots < a_{n_a} < s_1 < t_1 < s_2 < t_2 < \ldots < s_{n_s} < t_{n_s}$$
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. 7 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 $P_2$

$$\begin{array}{ll}
& a_1, a_2, s_1, t_1, s_2, t_2 \\
0 \xrightarrow{enq} 1 \to & (0, 1, 0, 0, 0, 1) \\
1 \xrightarrow{enq} 2 \to & (0, 1, 0, 1, 1, 0) \\
2 \xrightarrow{enq} 3 \to & (0, 1, 1, 1, 0, 1) \\
1 \xrightarrow{deq} 0 \to & (1, 0, 0, 0, 1, 0) \\
2 \xrightarrow{deq} 1 \to & (1, 0, 1, 0, 0, 1) \\
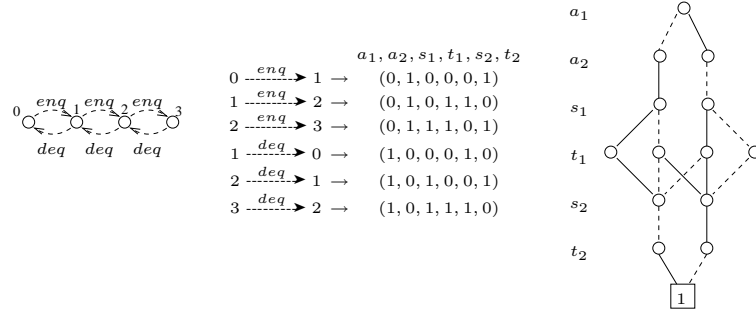3 \xrightarrow{deq} 2 \to & (1, 0, 1, 1, 1, 0)
\end{array}$$

**Fig. 7.** LTS, transition encoding and corresponding BDD for $Queue_0$

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:

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

The term on the first line is for the synchronising actions in which both $P_1$ and $P_2$ participate. The term on the second (third) 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.

We illustrate parallel composition by means of our queueing example. Fig. 8 shows the intermediate and final BDDs when performing BDD-based parallel composition of processes $Arrival$ and $Queue_0$. In the second (third) BDD one can observe the parts which express stability of process $Queue_0$ ($Arrival$). Even in this small example we observe the general tendency that the size of the resulting BDD (25 nodes, including the terminal false-node not shown) is in the order of the sum of the sizes of the two partner BDDs (15 nodes for $Queue_0$ and 8 nodes for $Arrival$, cf. Fig. 7 and Fig. 9). Thus, using BDD-based parallel composition, the typically observed exponential growth of memory requirements can be avoided.

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.

## 6.2 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 $a$
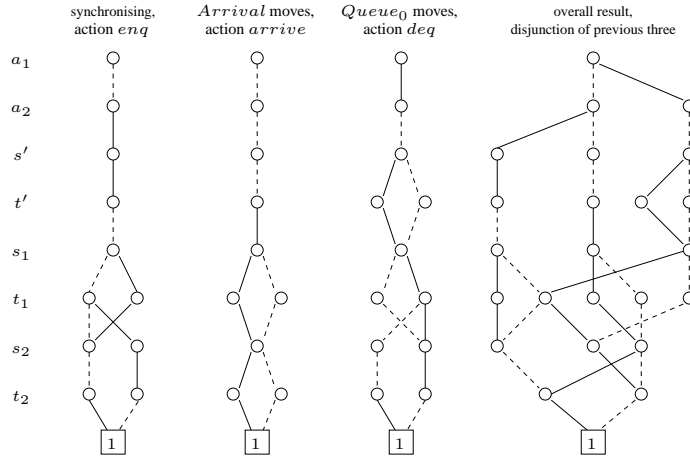
**Fig. 8.** Intermediate and final BDD results for parallel composition of $Arrival$ and $Queue_0$

(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.

### 6.3 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 modifications and augmentations 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 [8], edge-valued BDDs [26] and Binary Moment Diagrams (BMD) [7]. 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) [30]. 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 (for an example of a "don't care" see Fig. 10 below). Since these transitions are labelled by $2^k$ distinct rates, we need to assign a rate list of length
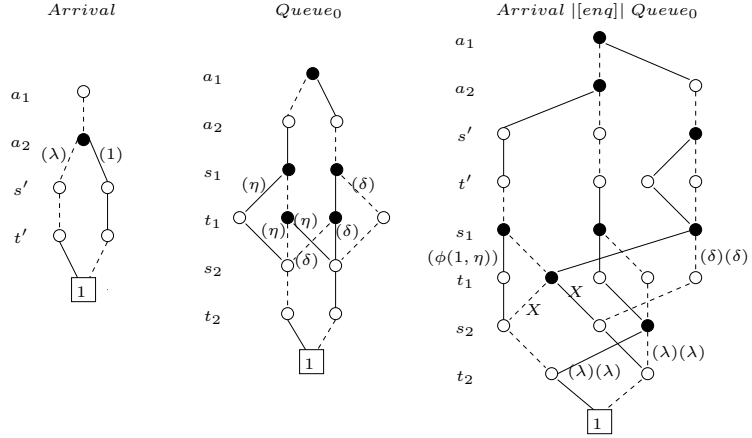
**Fig. 9.** DNBDDs for the queueing example (shorthand notation: $X = (\phi(1, \eta))(\delta)(\delta)$)

$2^k$ 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 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*

$$rates : Paths \rightarrow (I\!R)^+$$

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

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

*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. 9 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 $\lambda$ 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 hand side, 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.

**Fig. 10.** Encoded transitions with rates, and corresponding DNBDD with rate tree

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 [30]. In our current implementation of DNBDDs, the rate tree is implemented as illustrated in Fig. 10. 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 $(\lambda_0, \lambda_1)$ 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 and the rate lists are associated with the decision nodes of the BDD as indicated in Fig. 10 (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. We are currently investigating this issue.[2]

Parallel composition of two SLTSs based on their symbolic representation follows the same basic algorithm as sketched in Sec. 6.1. 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 $\phi$, any arithmetic expression of the two individual rates can be employed.

### 6.4 Symbolic Markovian bisimulation

We now discuss aspects of a DNBDD-based algorithm which computes Markovian bisimulation on SLTSs. The basic bisimulation algorithm is the same as in Sec. 4, only the procedure $split'$ needs to be adapted. When using DNBDDs, the cumulative rate

---

[2] In order to avoid such redundancies, an efficient data structure to represent rate trees might itself be based on BDDs.

of action $a$ from state $P$ to class $C_{spl}$ is computed in the following way: We compute $T_{P \xrightarrow{a} C_{spl}}(s,t)$, 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 $C(t)$):

$$T_{P \xrightarrow{a} C_{spl}}(s,t) := T_a(s,t) \wedge (s \doteq P) \wedge C_{spl}(t)$$

We use the notation $s \doteq P$ to denote that state $P$ is encoded as Boolean vector $s$. The cumulative rate $\gamma(P, a, C_{spl})$ is then computed by applying the function $sum\_of\_all\_rates$ to $T_{P \xrightarrow{a} C_{spl}}(s,t)$. This function simply sums up all the entries of all rate lists of a DNBDD. For example, application of the function $sum\_of\_all\_rates$ to the DNBDD in Fig. 10 yields $\lambda_0 + \lambda_1 + \alpha + \beta + \gamma + \delta$. Furthermore, in the $split\_tree$ used by procedure $split'$ (Fig. 4) the subclasses $C_{\gamma_1}, \ldots, C_{\gamma_k}$ are now also represented by BDDs.

**procedure** $split'(C, a, C_{spl}, Partition, Splitters)$
    **forall** $P \in C$
        $T_{P \xrightarrow{a} C_{spl}}(s,t) := T_a(s,t) \wedge (s \doteq P) \wedge C_{spl}(t)$
        $\gamma := sum\_of\_all\_rates(T_{P \xrightarrow{a} C_{spl}}(s,t))$
        /* 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** $(k > 1)$
        $\ldots$
        /* the remaining part of procedure $split'$ is as in Sec. 4, */
        /* but $Partition$ and $Splitters$ are represented as BDDs */

### 6.5 BDDs with and without rate information

The semantics of the complete language $\mathcal{L}$ comprises both types of transitions, action transitions $\dashrightarrow^{a}\dashrightarrow$ and Markovian transitions $\xrightarrow{a, \lambda}$, in one transition system, an ESLTS. Using the knowledge developed in the previous sections, an ESLTS can be encoded by means of two separate data structures, using BDDs to encode action transitions and DNBDDs to encode Markovian transitions. Also, during parallel composition, the component BDDs are treated separately from the component DNBDDs. Therefore the treatment of ESLTS does not pose specific problems. Furthermore, the computation of weak Markovian bisimilarity (Sec. 5) can be lifted to this combination of BDD and DNBDD. The computation of the weak transition relation $\Longrightarrow$ from $\dashrightarrow$ during the initialisation step can easily be performed on the BDD for the action transitions. Only the first part of function $split''$ requires the DNBDD information, in order to sort tangible markings in a $split\_tree$ (the tangibility predicate is encoded as a BDD as well), in analogy to the implementation of function $split'$ given in Sec. 6.4. The subsequent steps work completely on BDDs.

## 7 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, introducing DNBDDs to represent the additional rate information which is relevant for the analysis of the underlying Markov chain.

The complexity results established in this paper allow the following simple conclusion: the computational complexity of computing bisimulation equivalences *does not* increase when moving from a non-stochastic to a stochastic setting. For Markovian bisimilarity this fact is also mentioned (in similar settings) in [23] and in [2].

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 [5]. 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 actual relation between DNBDDs and MTBDDs, since MTBDD-based numerical analysis methods have already been developed [14, 21].

## References

1. C. Baier and H. Hermanns. Weak Bisimulation for Fully Probabilistic Processes. In *Proc. CAV '97*, Springer LNCS 1254:119–130, 1997.
2. M. Bernardo and R. Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science* 202:1-54, 1998.
3. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems* 14:25-59, 1987.
4. A. Bouali. Weak and branching bisimulation in FCTOOL. Rapports de Recherche 1575, INRIA Sophia Antipolis, Valbonne Cedex, France, 1992.

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

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

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

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

9. D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progressions. In *Proc. 19th ACM Symposium on Theory of Computing*, 1987.

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. J. van Glabbeek and W. Weijland:. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

13. J.F.Groote and F.W.Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. ICALP'90*, Springer LNCS 443:626-638, 1990.

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

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

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

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

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

19. H. Hermanns and M. Lohrey. Priority and maximal progress are completely axiomatisable. In *Proc. CONCUR'98*, Springer LNCS 1466:237-252, 1998.

20. H. Hermanns and M. Siegle. Computing Bisimulations for Stochastic Process Algebras using Symbolic Techniques. In *Proc. 6th Int. PAPM Workshop*, 103-118, Nice, 1998.

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

22. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

23. T. Huynh and L. Tian. On some Equivalence Relations for Probabilistic Processes. *Fundamenta Informaticae*, 17:211–234, 1992.

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

25. J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer, 1976.

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

27. K. Larsen and A. Skou. Bisimulation through Probabilistic Testing. *Information and Computation*, 94(1):1-28, 1991.

28. R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.

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

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