Multi-core symbolic bisimulation minimisation

Tom van Dijk* and Jaco van de Pol

Formal Methods and Tools, University of Twente, The Netherlands
{dijkt,vdpol}@cs.utwente.nl

Abstract. Bisimulation minimisation alleviates the exponential growth of transition systems in model checking by computing the smallest system that has the same behavior as the original system according to some notion of equivalence. One popular strategy to compute a bisimulation minimisation is signature-based partition refinement. This can be performed symbolically using binary decision diagrams to allow models with larger state spaces to be minimised.

This paper studies strong and branching symbolic bisimulation for labeled transition systems, continuous-time markov chains, and interactive markov chains. We introduce the notion of partition refinement with partial signatures. We extend the parallel BDD library Sylvan to parallelize the signature refinement algorithm, and develop a new parallel BDD algorithm to refine a partition, which conserves previous block numbers and uses a parallel data structure to store block assignments. We also present a specialized BDD algorithm for the computation of inert transitions. The experimental evaluation, based on benchmarks from the literature, demonstrates a speedup of up to 95x sequentially. In addition, we find parallel speedups of up to 17x due to parallelisation with 48 cores. Finally, we present the implementation of these algorithms as a versatile tool that can be customized for bisimulation minimisation in various contexts.

Keywords: multi-core, parallel, binary decision diagrams, bisimulation minimisation, labeled transition systems, continuous-time markov chains, interactive markov chains

1 Introduction

One core challenge in model checking is the state space explosion problem. The space and time requirements of model checking increase exponentially with the size of the models. Bisimulation minimisation computes the smallest equivalent model (maximal bisimulation) under some notion of equivalence, which can significantly reduce the number of states. This technique is also used to abstract models from internal behavior, when only observable behavior is relevant.

The maximal bisimulation of a model is typically computed using partition refinement. Starting with an initially coarse partition (e.g. all states are equivalent), the partition is refined until states in each equivalence class can no longer

* The first author is supported by the NWO project MaDriD, grant nr. 612.001.101

be distinguished. The result is the maximal bisimulation with respect to the initial partition. Blom et al. [3] introduced a signature-based method for partition refinement, which assigns states to equivalence classes according to a characterizing signature. This method easily extends to various types of bisimulation.

Another well-known method to deal with very large state spaces is symbolic model checking, where sets of states are represented by their characteristic function, which is efficiently stored using binary decision diagrams (BDDs). In the literature, symbolic methods have been applied to bisimulation minimisation in several ways. Bouali and De Simone [5] refine the equivalence relation $R \subseteq S \times S$, by iteratively removing all "bad" pairs from R, i.e., pairs of states that are no longer equivalent. For strong bisimulation, Mumme and Ciardo [19] apply saturation-based methods to compute R. Wimmer et al. [25,26] use signatures to refine the partition, represented by the assignment to equivalence classes $P \colon S \to C$. Symbolic bisimulation based on signatures has also been applied to Markov chains by Derisavi [11] and Wimmer et al. [23,24].

The symbolic representation of the maximal bisimulation, when effective, often tends to be much larger than the original model. One particular application of symbolic bisimulation minimisation is as a bridge between symbolical models and explicit-state analysis algorithms. Such models can have very large state spaces that are efficiently encoded using BDDs. If the minimised model is sufficiently small, then it can be analyzed efficiently using explicit-state algorithms.

These techniques mainly reduce the memory requirements of model checking. To take advantage of computer systems with multiple processors, developing scalable parallel algorithms is the way forward. In [12,14], we implemented the multi-core BDD package Sylvan, applying parallelism to symbolic model checking. Parallelization has also been applied to explicit-state bisimulation minimisation. Blom et al. [2,3] introduced a parallel, signature-based algorithm for various types of bisimulation, especially strong and branching bisimulation. Also, [17] proposed a concurrent algorithm for bisimulation minimisation which combines signatures with the approach by Paige and Tarjan [20]. Recently, Wijs [22] implemented highly parallel strong and branching bisimilarity checking on GPGPUs. As far as we are aware, no earlier work combines symbolic bisimulation minimisation and parallelism.

In the current paper, we study bisimulation minimisation for labeled transition systems (LTSs), continuous-time Markov chains (CTMCs) and interactive Markov chains (IMCs), which combines the features of LTSs and CTMCs. These allow the analysis of quantitative properties, e.g. performance and dependability.

We concentrate on strong bisimulation and branching bisimulation. Strong bisimulation preserves both internal behavior (τ-transitions) and observable behavior, while branching bisimulation abstracts from internal behavior. The advantage of branching bisimulation compared to other variations of weak bisimulation is that it preserves the branching structure of the LTS, thus preserving certain interesting properties such as CTL* without next-state operator [9].

The current paper contains the following contributions. We introduce the notion of partition refinement with partial signatures in Section 3. Section 4 discusses how we extend the multi-core BDD package Sylvan to parallelize signature-based partition refinement. In particular, we develop two specialized BDD algorithms. We implement a new `refine` algorithm, that refines a partition according to a signature, but maximally reuses the block number assignment of the previous partition (Section 4.3). This algorithm improves the operation cache use for the computation of the signatures of stable blocks, and enables partition refinement with partial signatures. We also present the `inert` algorithm, which, given a transition relation and a partition, removes all transitions that are not inert (Section 4.4). This algorithm avoids an expensive intermediate result reported in the literature [26]. Section 5 discusses experimental data based on benchmarks from the literature to demonstrate a speedup of up to 95x sequentially. In addition, we find parallel speedups of up to 17x due to parallelisation with 48 cores. Finally, we present the implementation of these algorithms as a versatile tool that can be customized for bisimulation minimisation in various contexts.

2 Preliminaries

We recall the basic definitions of partitions, of LTSs, of CTMCs, of IMCs, and of various bisimulations as in [3,15,25,26,27].

Definition 1. *Given a set S, a partition π of S is a subset $\pi \subseteq 2^S$ such that*

$$\bigcup_{C \in \pi} C = S \qquad \textit{and} \qquad \forall C, C' \in \pi \colon (C = C' \vee C \cap C' = \emptyset).$$

If π' and π are two partitions, then π' is a refinement of π, written $\pi' \sqsubseteq \pi$, if each block of π' is contained in a block of π. The elements of π are called equivalence classes or blocks. Each equivalence relation $\equiv$ is associated with a partition $\pi = S/\equiv$. In this paper, we use π and $\equiv$ interchangeably.

Definition 2. *A labeled transition system (LTS) is a tuple $(S, \mathsf{Act}, \rightarrow)$, consisting of a set of states S, a set of labels Act that may contain the non-observable action τ, and transitions $\rightarrow \subseteq S \times \mathsf{Act} \times S$.*

We write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. and $s \overset{\tau}{\nrightarrow}$ when s has no outgoing τ-transitions. We use $\xrightarrow{a*}$ to denote the transitive reflexive closure of $\xrightarrow{a}$. Given an equivalence relation $\equiv$, we write $\xrightarrow[\equiv]{a}$ for $\xrightarrow{a} \cap \equiv$, i.e., transitions between equivalent states, called *inert* transitions. We use $\xrightarrow[\equiv]{a*}$ for the transitive reflexive closure of $\xrightarrow[\equiv]{a}$.

Definition 3. *A continuous-time Markov chain (CTMC) is a tuple $(S, \Rightarrow)$, consisting of a set of states S and Markovian transitions $\Rightarrow \subseteq S \times \mathbb{R}^{>0} \times S$.*

We write $s \overset{\lambda}{\Rightarrow} t$ for $(s, \lambda, t) \in \Rightarrow$. The interpretation of $s \overset{\lambda}{\Rightarrow} t$ is that the CTMC can switch from s to t within d time units with probability $1 - e^{-\lambda \cdot d}$. For a state s, let $\mathbf{R}(s)(s') = \sum\{\lambda \mid s \overset{\lambda}{\Rightarrow} s'\}$ be the rate to move from state s to state s', and let $\mathbf{R}(s)(C) = \sum_{s' \in C} \mathbf{R}(s)(s')$ be the cumulative rate to reach a set of states $C \subseteq S$ from state s.

Definition 4. *An interactive Markov chain (IMC) is a tuple $(S, \mathsf{Act}, \rightarrow, \Rightarrow)$, consisting of a set of states S, a set of labels Act that may contain the non-observable action τ, transitions $\rightarrow \subseteq S \times \mathsf{Act} \times S$, and Markovian transitions $\Rightarrow \subseteq S \times \mathbb{R}^{>0} \times S$.*

An IMC basically combines the features of an LTS and a CTMC. One feature of IMCs is the *maximal progress assumption.* Internal interactive transitions, i.e. τ-transitions, can be assumed to take place immediately, while the probability that a Markovian transition executes immediately is zero. Therefore, we may remove all Markovian transitions from states that have outgoing τ-transitions: $s \xrightarrow{\tau}$ implies $\mathbf{R}(s)(S) = 0$. We call IMCs to which this operation has been applied *maximal-progress-cut* (mp-cut) IMCs.

For LTSs, strong and branching bisimulation are typically defined as follows [26]:

Definition 5. *An equivalence relation $\equiv_S$ is a strong bisimulation on an LTS if for all states s, t, s' with $s \equiv_S t$ and for all $s \xrightarrow{a} s'$, there exists a state t' with $t \xrightarrow{a} t'$ and $s' \equiv_S t'$.*

Definition 6. *An equivalence relation $\equiv_B$ is a branching bisimulation on an LTS if for all states s, t, s' with $s \equiv_B t$ and for all $s \xrightarrow{a} s'$, either*

- *$a = \tau$ and $s' \equiv_B t$, or*
- *there exist states t', t'' with $t \xrightarrow{\tau *} t' \xrightarrow{a} t''$ and $t \equiv_B t'$ and $s' \equiv_B t''$.*

For CTMCs, strong bisimulation is defined as follows [11,23]:

Definition 7. *An equivalence relation $\equiv_S$ is a strong bisimulation on a CTMC if for all $(s, t) \in \equiv_S$ and for all classes $C \in S/\equiv_S$, $\boldsymbol{R}(s)(C) = \boldsymbol{R}(t)(C)$.*

For mp-cut IMCs, strong and branching bisimulation are defined as follows [15,27]:

Definition 8. *An equivalence relation $\equiv_S$ is a strong bisimulation on an mp-cut IMC if for all $(s, t) \in\equiv_S$ and for all classes $C \in S/\equiv_S$*

- *$s \xrightarrow{a} s'$ for some $s' \in C$ implies $t \xrightarrow{a} t'$ for some $t' \in C$*
- *$\boldsymbol{R}(s)(C) = \boldsymbol{R}(t)(C)$*

Definition 9. *An equivalence relation $\equiv_B$ is a branching bisimulation on an mp-cut IMC if for all $(s, t) \in\equiv_B$ and for all classes $C \in S/\equiv_B$*

- *$s \xrightarrow{a} s'$ for some $s' \in C$ implies*
  - *$a = \tau$ and $(s, s') \in \equiv_B$, or*
  - *there exist states $t', t'' \in S$ with $t \xrightarrow{\tau *} t' \xrightarrow{a} t''$ and $(t, t') \in \equiv_B$ and $t'' \in C$.*
- *$\boldsymbol{R}(s)(C) > 0$ implies*
  - *$\boldsymbol{R}(s)(C) = \boldsymbol{R}(t')(C)$ for some $t' \in S$ such that $t \xrightarrow{\tau *} t' \overset{\tau}{\not\rightarrow}$ and $(t, t') \in \equiv_B$.*
- *$s \overset{\tau}{\not\rightarrow}$ implies $t \xrightarrow{\tau *} t' \overset{\tau}{\not\rightarrow}$ for some t'*

3 Signature-based Bisimulation

Blom and Orzan [3] introduced a signature-based approach to compute the maximal bisimulation of an LTS, which was further developed into a symbolic method by Wimmer et al. [26]. Each state is characterized by a *signature*, which is the same for all equivalent states in a bisimulation. These signatures are used to refine a partition of the state space until a fixed point is reached, which is the maximal bisimulation.

In the literature, multiple signatures are sometimes used that together fully characterize states, for example based on the state labels, based on the rates of continous-time transitions, and based on the enabled interactive transitions. In the current paper, these multiple signatures are considered elements of a single signature that fully characterizes each state.

Definition 10. *A signature $\sigma(\pi)(s)$ is a tuple of functions $f_i(\pi)(s)$, that together characterize each state s with respect to a partition π.*

Two signatures $\sigma(\pi)(s)$ and $\sigma(\pi)(t)$ are equivalent, if and only if for all f_i, $f_i(\pi)(s) = f_i(\pi)(t)$.

The signatures of five bisimulations from Section 2 are known from the literature. For all actions $a \in \mathsf{Act}$ and equivalence classes $C \in \pi$, we define

- $\mathbf{T}(\pi)(s) = \{(a, C) \mid \exists s' \in C \colon s \xrightarrow{a} s'\}$
- $\mathbf{B}(\pi)(s) = \{(a, C) \mid \exists s' \in C \colon s \xrightarrow[\pi]{\tau*} \xrightarrow{a} s' \wedge \neg(a = \tau \wedge s \in C)\}$
- $\mathbf{R}^s(\pi)(s) = C \mapsto \mathbf{R}(s)(C)$
- $\mathbf{R}^b(\pi)(s) = C \mapsto \max(\{\mathbf{R}(s')(C) \mid \exists s' \colon s \xrightarrow[\pi]{\tau*} s' \overset{\tau}{\nrightarrow}\})$

The five bisimulations are associated with the following signatures:

Strong bisimulation for an LTS	$(\mathbf{T})$	[26]
Branching bisimulation for an LTS	$(\mathbf{B})$	[26]
Strong bisimulation for a CTMC	$(\mathbf{R}^s)$	[23]
Strong bisimulation for an mp-cut IMC	$(\mathbf{T}, \mathbf{R}^s)$	[27]
Branching bisimulation for an mp-cut IMC	$(\mathbf{B}, \mathbf{R}^b, s \xrightarrow{\tau*} \overset{\tau}{\nrightarrow})$	[27]

Functions $\mathbf{T}$ and $\mathbf{B}$ assign to each state s all actions a and equivalence classes $C \in \pi$, such that state s can reach C by an action a either directly $(\mathbf{T})$ or via any number of inert τ-steps $(\mathbf{B})$. $\mathbf{R}^s$ equals $\mathbf{R}$ but with the domain restricted to the equivalence classes $C \in \pi$, and represents the cumulative rate with which state s can go to states in C. $\mathbf{R}^b$ equals $\mathbf{R}^s$ for states $s \overset{\tau}{\nrightarrow}$, and takes the highest "reachable rate" for states with inert τ-transitions. In branching bisimulation for mp-cut IMCs, the "highest reachable rate" is by definition the rate that all states $s \overset{\tau}{\nrightarrow}$ in C have. The element $s \xrightarrow{\tau*} \overset{\tau}{\nrightarrow}$ distinguishes time-convergent states from time-divergent states [27], and is independent of the partition.

For the bisimulations of Definitions 5–9, we state:

Lemma 1. *A partition π is a bisimulation, if and only if for all s and t that are equivalent in π, $\sigma(\pi)(s) = \sigma(\pi)(t)$.*

For the above definitions it is fairly straightforward to prove that they are equivalent to the classical definitions of bisimulation. See e.g. [3,26] for the bisimulations on LTSs and [27] for the bisimulations on IMCs.

3.1 Partition refinement

The definition of signature-based partition refinement is as follows.

Definition 11 (Partition refinement with full signatures).

$$\begin{aligned}\operatorname{sigref}(\pi, \sigma) &:= \{\{t \in S \mid \sigma(\pi)(s) = \sigma(\pi)(t)\} \mid s \in S\} \\ \pi^0 &:= \{S\} \\ \pi^{n+1} &:= \operatorname{sigref}(\pi^n, \sigma)\end{aligned}$$

The algorithm iteratively refines the initial coarsest partition $\{S\}$ according to the signatures of the states, until some fixed point $\pi^{n+1} = \pi^n$ is obtained. This fixed point is the maximal bisimulation for "monotone signatures":

Definition 12. *A signature is monotone if for all π, π' with $\pi \sqsubseteq \pi'$, whenever $\sigma(\pi)(s) = \sigma(\pi)(t)$, also $\sigma(\pi')(s) = \sigma(\pi')(t)$.*

For all monotone signatures, the sigref operator is monotone: $\pi \sqsubseteq \pi'$ implies $\operatorname{sigref}(\pi, \sigma) \sqsubseteq \operatorname{sigref}(\pi', \sigma)$. Hence, following Kleene's fixed point theorem, the procedure above reaches the greatest fixed point.

In Definition 11, the full signature is computed in every iteration. We propose to apply partition refinement using parts of the signature. By definition, $\sigma(\pi)(s) = \sigma(\pi)(t)$ if and only if for all parts $f_i(\pi)(s) = f_i(\pi)(t)$.

Definition 13 (Partition refinement with partial signatures).

$$\begin{aligned}\operatorname{sigref}(\pi, f_i) &:= \{\{t \in S \mid f_i(\pi)(s) = f_i(\pi)(t) \wedge s \equiv_\pi t\} \mid s \in S\} \\ \pi^0 &:= \{S\} \\ \pi^{n+1} &:= \operatorname{sigref}(\pi^n, f_i) \qquad (\text{select } f_i \in \sigma)\end{aligned}$$

We always select some f_i that refines the partition π. A fixed point is reached only when no f_i refines the partition further: $\forall f_i \in \sigma\colon \operatorname{sigref}(\pi^n, f_i) = \pi^n$. The extra clause $s \equiv_\pi t$ ensures that every application of sigref refines the partition.

Theorem 1. *If all parts f_i are monotone, Def. 13 yields the greatest fixed point.*

Proof. The procedure terminates since the chain is decreasing ($\pi^{n+1} \sqsubseteq \pi^n$), due to the added clause $s \equiv_\pi t$. We reach some fixed point π^n, since $\forall f_i \in \sigma\colon \operatorname{sigref}(\pi^n, f_i) = \pi^n$ implies $\operatorname{sigref}(\pi^n, \sigma) = \pi^n$. Finally, to prove that we get the *greatest* fixed point, assume there exists another fixed point $\xi = \operatorname{sigref}(\xi, \sigma)$. Then also $\xi = \operatorname{sigref}(\xi, f_i)$ for all i. We prove that $\xi \sqsubseteq \pi^n$ by induction on n. Initially, $\xi \sqsubseteq S = \pi^0$. Assume $\xi \sqsubseteq \pi^n$, then for the selected i, $\xi = \operatorname{sigref}(\xi, f_i) \sqsubseteq \operatorname{sigref}(\pi^n, f_i) = \pi^{n+1}$, using monotonicity of f_i. □

There are several advantages to this approach due to its flexibility. First, for any f_i that is independent of the partition, refinement with respect to that f_i only needs to be applied once. Furthermore, refinements can be applied according to different strategies. For instance, for the strong bisimulation of an mp-cut IMC, one could refine w.r.t. $\mathbf{T}$ until there is no more refinement, then w.r.t. $\mathbf{R}^s$ until there is no more refinement, then repeat until neither $\mathbf{T}$ nor $\mathbf{R}^s$ refines the partition. Finally, computing the full signature is the most memory-intensive operation in symbolic signature-based partition refinement. If the partial signatures are smaller than the full signature, then larger models can be minimised.

4 Symbolic signature refinement

This section describes the parallel decision diagram library Sylvan, followed by the (MT)BDDs and (MT)BDD operations required for signature-based partition refinement. We describe how we encode partitions and signatures for signature-based partition refinement. We present a new parallelized `refine` function that maximally reuses block numbers from the old partition. Finally, we present a new BDD algorithm that computes inert transitions, i.e., restricts a transition relation such that states s and s' are in the same block.

4.1 Decision diagram algorithms in Sylvan

In symbolic model checking [7], sets of states and transitions are represented by their characteristic function, rather than stored individually. With states described by N Boolean variables, a set $S \subseteq \mathbb{B}^N$ can be represented by its characteristic function $f\colon \mathbb{B}^N \to \mathbb{B}$, where $S = \{s \mid f(s)\}$. Binary decision diagrams (BDDs) are a concise and canonical representation of Boolean functions [6].

An (ordered) BDD is a directed acyclic graph with leaves 0 and 1. Each internal node has a variable label x_i and two outgoing edges labeled 0 and 1. Variables are encountered along each path according to a fixed variable ordering. Duplicate nodes and nodes with two identical outgoing edges are forbidden. It is well known that for a fixed variable ordering, every Boolean function is represented by a unique BDD.

In addition to BDDs with leaves 0 and 1, multi-terminal binary decision diagrams have been proposed [1,8] with leaves other than 0 and 1, representing functions from the Boolean space $\mathbb{B}^N$ onto any finite set. For example, MTBDDs can have leaves representing integers (encoding $\mathbb{B}^N \to \mathbb{N}$), floating-point numbers (encoding $\mathbb{B}^N \to \mathbb{R}$) and rational numbers (encoding $\mathbb{B}^N \to \mathbb{Q}$). Partial functions are supported using a terminal leaf $\bot$.

Sylvan [12,14] implements parallelized operations on decision diagrams using parallel data structures and work-stealing. Work-stealing [4,13] is a load balancing method for task-based parallelism. Recursive operations, such as most BDD operations, implicitly form a tree of tasks. Independent subtasks are stored in queues and idle processors steal tasks from the queues of busy processors.

```
def apply($x$, $y$, F):
    if $(x, y, \mathsf{F}) \in$ cache : return cache$[(x, y, \mathsf{F})]$          /* get from cache */
    if $x$ and $y$ are terminals : return $\mathsf{F}(x, y)$           /* apply operator F */
    $v$ = topVar($x$,$y$)
    low ← apply($x_{v=0}$, $y_{v=0}$, F)                        /* execute in parallel */
    high ← apply($x_{v=1}$, $y_{v=1}$, F)
    result ← BDDnode($v$, low, high)                                /* compute result */
    cache$[(x, y, \mathsf{F})]$ ← result                                   /* put in cache */
    return result
```

Algorithm 1: Generic algorithm that applies a binary operator F to BDDs x and y.

Algorithm 1 describes the implementation of a generic binary operation F. BDD operations mainly consist of consulting an operation cache, performing some recursive step, and creating new BDD nodes using a unique table. The operation cache is required to reduce the time complexity of BDD operations from exponential to polynomial in the size of the BDDs. Sylvan uses a single shared unique table for all BDD nodes and a single shared operation cache for all operations. To obtain high performance in a multi-core environment, the datastructures for the BDD nodes and the operation cache must be highly scalable. Sylvan implements several non-blocking datastructures to enable good speedups [14].

To compute symbolic signature-based partition refinement, several basic operations must be supported by the BDD package (see also [26]). Sylvan implements basic operations such as $\wedge$ and `if-then-else`, and existential quantification $\exists$. Negation $\neg$ is performed in constant time using complement edges. To compute relational products of transition systems, there are operations `relnext` (to compute successors) and `relprev` (to compute predecessors and to concatenate relations), which combine the relational product with variable renaming. Similar operations are also implemented for MTBDDs. Sylvan is designed to support custom BDD algorithms. We present two new algorithms below.

4.2 Encoding of signature refinement

We implement symbolic signature refinement similar to [26]. Unlike [26], we do not refine the partition with respect to a single block, but with respect to all blocks simultaneously. We use a binary encoding with variables s for the current state, s' for the next state, a for the action labels and b for the blocks. We order BDD variables a and b after s and s', since this is required to efficiently replace signatures (a, b) by new block numbers b (see below). Variables s and s' are interleaved, which is common in the context of transition systems.

To perform symbolic bisimulation we represent a number of sets by their characteristic functions. See also Figure 1.

- A set of states is represented by a BDD $\mathcal{S}(s)$;
- Transitions are represented by a BDD $\mathcal{T}(s, s', a)$;
- Markovian transitions are represented by an MTBDD $\mathcal{R}(s, s')$, with leaves containing rational numbers ($\mathbb{Q}$);

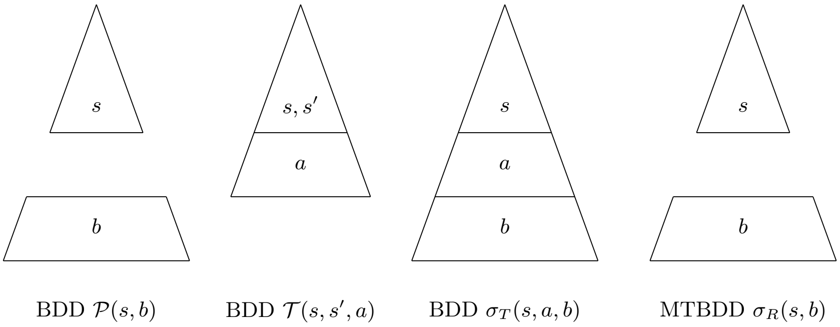

Fig. 1. Schematic overview of the BDDs in signature refinement

- Signatures $\mathbf{T}$ and $\mathbf{B}$ are represented by a BDD $\sigma_T(s,a,b)$;
- Signatures $\mathbf{R}^s$ and $\mathbf{R}^b$ are represented by an MTBDD $\sigma_R(s,b)$.

In the literature, three methods have been proposed to represent π.

1. As an equivalence relation, using a BDD $\mathcal{E}(s,s') = 1$ iff $s \equiv_\pi s'$ [5,19].
2. As a partition, by assigning each block a unique number, encoded with variables b, using a BDD $\mathcal{P}(s,b) = 1$ iff $s \in C_b$ [11,26,27].
3. Using $k = \lceil \log_2 n \rceil$ BDDs $\mathcal{P}_0, \ldots, \mathcal{P}_{k-1}$ such that $\mathcal{P}_i(s) = 1$ iff $s \in C_b$ and the i^{th} bit of b is 1. This requires significant time to restore blocks for the refinement procedure, but can require less memory [10].

We choose to use method 2, since in practice the BDD of $\mathcal{P}(s,b)$ is smaller than the BDD of $\mathcal{E}(s,s')$. Using $\mathcal{P}(s,b)$ also has the advantage of straight-forward signature computation. The logarithmic representation is incompatible with our approach, since we refine all blocks simultaneously. Their approach involves restoring individual blocks to the $\mathcal{P}(s,b)$ representation, performing a refinement step, and compacting the result to the logarithmic representation. Restoring all blocks simply computes the full $\mathcal{P}(s,b)$.

We represent Markovian transitions using rational numbers, since they offer better precision than floating-point numbers. The manipulation of floating-point numbers typically introduces tiny rounding errors, resulting in different results of similar computations. This significantly affects bisimulation reduction, often resulting in finer partitions than the maximal bisimulation [23].

4.3 The `refine` algorithm

In this section, we present a new BDD algorithm to refine partitions according to a signature, which maximally preserves previously assigned block numbers.

Partition refinement consists of two steps: computing the signatures and computing the next partition. Given the signatures σ_T and/or σ_R for the current partition π, the new partition can be computed as follows.

```
def refine($\sigma$, $\mathcal{P}$):
    if $(\sigma, \mathcal{P}, \text{iter}) \in$ cache : return cache[$(\sigma, \mathcal{P}, \text{iter})$]
    $v$ = topVar($\sigma$, $\mathcal{P}$)
    if $v$ equals $s_i$ for some $i$ :
        # match paths on $s$ in $\sigma$ and $\mathcal{P}$
        low ← refine($\sigma_{s_i=0}$, $\mathcal{P}_{s_i=0}$)
        high ← refine($\sigma_{s_i=1}$, $\mathcal{P}_{s_i=1}$)
        result ← BDDnode($s_i$, low, high)
    else:
        # $\sigma$ now encodes the state signature
        # $\mathcal{P}$ now encodes the previous block
        $B$ ← decodeBlock($\mathcal{P}$)
        # try to claim block B if still free
        if blocks[$B$].sig $= \bot$ : cas(blocks[$B$].sig, $\bot$, $\sigma$)
        if blocks[$B$].sig $= \sigma$ : result ← $\mathcal{P}$
        else:
            $B$ ← search_or_insert($\sigma$, $B$)
            result ← encodeBlock($B$)
    cache[$(\sigma, \mathcal{P}, \text{iter})$] ← result
    return result
```

Algorithm 2: `refine`, the (MT)BDD operation that assigns block numbers to signatures, given a signature σ and the previous partition $\mathcal{P}$.

Since the chosen variable ordering has variables s, s' before a, b, each path in σ ends in a (MT)BDD representing the signature for the states encoded by that path. For σ_T, every path that assigns values to s ends in a BDD on a, b. For σ_R, every path that assigns values to s ends in a MTBDD on b with rational leaves.

Wimmer et al. [26] present a BDD operation `refine` that "replaces" these sub-(MT)BDDs by the BDD representing a unique block number for each distinct signature. The result is the BDD of the next partition. They use a global counter and a hash table to associate each signature with a unique block number. This algorithm has the disadvantage that block number assignments are unstable. There is no guarantee that a stable block has the same block number in the next iteration. This has implications for the computation of the new signatures. When the block number of a stable block changes, cached results of signature computation in earlier iterations cannot be reused.

We modify the `refine` algorithm to use the current partition to reuse the previous block number of each state. This also allows refining a partition with respect to only a part of the signature, as described in Section 3. The modification is applied such that it can be parallelized in Sylvan. See Algorithm 2.

The algorithm has two input parameters: the (MT)BDD σ which encodes the (partial) signature for the current partition, and the BDD $\mathcal{P}$ which encodes the current partition. The algorithm uses a global counter iter, which is the current iteration of partition refinement. This is necessary since the cached results of the previous iteration cannot be reused. It also uses and updates an array `blocks`,

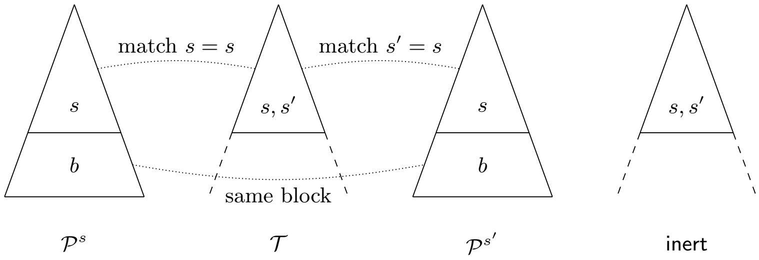

Fig. 2. Schematic overview of the BDDs in the `inert` algorithm

which contains the signature of each block in the new partition. This array is cleared between iterations of partition refinement.

The implementation is similar to other BDD operations, featuring the use of the operation cache (lines 2 and 15) and a recursion step for variables in s (lines 3–7), with the two recursive operations executed in parallel. `refine` simultaneously descends in σ and $\mathcal{P}$ (lines 5–6), matching the valuation of s_i in σ and $\mathcal{P}$. Block assignment happens at lines 9–14. We rely on the well-known atomic operation `compare_and_swap` (`cas`), which atomically compares and modifies a value in memory. This is necessary so the algorithm is still correct when parallelized. We use `cas` to claim a block number for the signature (line 10). If the block number is already used for a different signature, then this block is being refined and we call a method `search_or_insert` to assign a new block number.

Different implementations of `search_and_insert` are possible. We implemented a parallel hash table that uses a global counter for the next block number when inserting a new pair (σ, B), similar to [26]. An alternative implementation that performed better in our experiments integrates the `blocks` array with a skip list. A skip list is a probabilistic multi-level ordered linked list. See [21].

4.4 Computing inert transitions

To compute the set of inert τ-transitions for branching bisimulation, i.e., $s \xrightarrow[\pi]{\tau} s'$, or more generally, to compute any inert transition relation $\rightarrow \cap \equiv$ where $\equiv$ is the equivalence relation corresponding to π computed by $\mathcal{E}(s, s') = \exists b \colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$, the expression $\mathcal{T}(s, s') \wedge \exists b \colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$ must be computed. [26] writes that the intermediate BDD of $\exists b \colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$, obtained by first computing $\mathcal{P}(s', b)$ using variable renaming from $\mathcal{P}(s, b)$ and then $\exists b \colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$ using `and_exists`, is very large. This makes sense, since this intermediate result is indeed the BDD $\mathcal{E}(s, s')$, which we were avoiding by representing the partition using $\mathcal{P}(s, b)$.

The solution in [26] was to avoid computing $\mathcal{E}$ by computing the signatures and the refinement only with respect to one block at a time, which also enables several optimizations in [25].

```
def inert($\mathcal{T}$, $\mathcal{P}^s$, $\mathcal{P}^{s'}$):
    if $(\mathcal{T}, \mathcal{P}^s, \mathcal{P}^{s'}) \in$ cache : return cache$[(\mathcal{T}, \mathcal{P}^s, \mathcal{P}^{s'})]$
    # find highest variable, interpreting $s_i$ in $\mathcal{P}^{s'}$ as $s'_i$
    $v$ = topVar($\mathcal{T}$, $\mathcal{P}^s$, $\mathcal{P}^{s'}$)
    if $v$ equals $s_i$ for some $i$ :
        # match $s_i$ in $\mathcal{T}$ with $\mathcal{P}^s$
        low ← inert($\mathcal{T}_{s_i=0}$, $\mathcal{P}^s_{s_i=0}$, $\mathcal{P}^{s'}$)
        high ← inert($\mathcal{T}_{s_i=1}$, $\mathcal{P}^s_{s_i=1}$, $\mathcal{P}^{s'}$)
        result ← BDDnode($s_i$, low, high)
    elif $v$ equals $s'_i$ for some $i$ :
        # match $s'_i$ in $\mathcal{T}$ with $s_i$ in $\mathcal{P}^{s'}$
        low ← inert($\mathcal{T}_{s'_i=0}$, $\mathcal{P}^s$, $\mathcal{P}^{s'}_{s_i=0}$)
        high ← inert($\mathcal{T}_{s'_i=1}$, $\mathcal{P}^s$, $\mathcal{P}^{s'}_{s_i=1}$)
        result ← BDDnode($s'_i$, low, high)
    else:
        # match the blocks $\mathcal{P}^s$ and $\mathcal{P}^{s'}$
        if $\mathcal{P}^s \neq \mathcal{P}^{s'}$ : result ← False
        else: result ← $\mathcal{T}$
    cache$[(\mathcal{T}, \mathcal{P}^s, \mathcal{P}^{s'})]$ ← result
    return result
```

Algorithm 3: Computes the inert transitions of a transition relation $\mathcal{T}$ according to the block assignments to current states ($\mathcal{P}^s$) and next states ($\mathcal{P}^{s'}$).

We present an alternative solution, which computes $\rightarrow \cap \equiv$ directly using a custom BDD algorithm. The `inert` algorithm takes parameters $\mathcal{T}(s, s')$ ($\mathcal{T}$ may contain other variables ordered after s, s') and two copies of $\mathcal{P}(s, b)$: $\mathcal{P}^s$ and $\mathcal{P}^{s'}$. The algorithm matches $\mathcal{T}$ and $\mathcal{P}^s$ on valuations of variables s, and $\mathcal{T}$ and $\mathcal{P}^{s'}$ on valuations of variables s'. See Algorithm 3, and also Figure 2 for a schematic overview. When in the recursive call all valuations to s and s' have been matched, with $S_s, S_{s'} \subseteq S$ the sets of states represented by these valuations, then $\mathcal{T}$ is the set of actions that label the transitions between states in S_s and $S_{s'}$, $\mathcal{P}^s$ is the block that contains all S_s and $\mathcal{P}^{s'}$ is the block that contains all $S_{s'}$. Then if $\mathcal{P}^s \neq \mathcal{P}^{s'}$, the transitions are not inert and `inert` returns `False`, removing the transition from $\mathcal{T}$. Otherwise, $\mathcal{T}$ (which may still contain other variables ordered after s, s', such as action labels), is returned.

5 Experimental evaluation

5.1 Tool support

We implemented multi-core signature-based partition refinement in a tool called SigrefMC, using the (MT)BDD-package Sylvan [12,14].

The tool supports LTSs, CTMCs and IMCs delivered in two input formats, the XML format used by the original Sigref tool, and the BDD format that the tool LTSMIN [16] generates for various model checking languages. SIGREFMC supports both the floating-point and the rational representation of rates in continuous-time transitions.

One of the design goals of this tool is to encourage researchers to extend it for their own file formats and notions of bisimulation, and to integrate it in other toolsets. Therefore, SIGREFMC is freely available online[1] and licensed with the MIT license. Documentation is available and instructions for extending the tool for different input/output formats and types of bisimulation are included.

5.2 Experiments

To study the improvements presented in the current paper, we compared our results (using the skip list variant of `refine`) to SIGREF 1.5 [25] for LTS and IMC models, and to a version of SIGREF used in [23] for CTMC models. For the CTMC models, we used SIGREF with rational numbers provided by the GMP library and SIGREFMC with rational number support by Sylvan. For the IMC models, version 1.5 of SIGREF does not support the GMP library and the version used in [23] does not support IMCs. We used SIGREFMC with floating points for a fairer comparison, but the tools give a slightly different number of blocks.

In the current paper, we restrict ourselves to the models presented in [23,26] and an IMC model that is part of the distribution of SIGREF. These models have been generated from PRISM benchmarks using a custom version of the PRISM toolset [18]. We refer to the literature for a description of these models.

We perform experiments on the three tools using the same 48-core machine, containing 4 AMD Opteron$^{\text{TM}}$ 6168 processors with 12 cores each. We measure the runtimes for partition refinement using SIGREF, SIGREFMC with only 1 worker, and SIGREFMC with 48 workers.

Note that apart from the new `refine` and `inert` algorithms presented in the current paper, there are several other differences. The first is that the original SIGREF uses the CUDD implementation of BDDs, while SIGREFMC obviously uses Sylvan, along with some extra BDD algorithms that avoid explicitly computing variable renaming of some BDDs. The second is that SIGREF has several optimizations [25] that are not available in SIGREFMC.

5.3 Results

See Table 1 for the results of these experiments. These results were obtained by repeating each benchmark at least 15 times and taking the average. The timeout was set to 3600 seconds. The column “States” shows the number of states before bisimulation minimisation, and “Blocks” the number of equivalence classes after bisimulation minimisation. We show the wallclock time using SIGREF (T_w), using SIGREFMC with 1 worker (T_1) and using SIGREFMC with 48 workers (T_{48}). We compute the sequential speedup T_w/T_1, the parallel speedup T_1/T_{48} and the total speedup T_w/T_{48}.

[1] https://github.com/utwente-fmt/sigrefmc

Due to space constraints, we do not include all results, but restrict ourselves to larger models. We refer to the full experimental data that is available online[2]. In the full set of results, excluding executions that take less than 1 second, SIGREFMC is always faster sequentially and always benefits from parallelism.

The results show a clear advantage for larger models. One interesting result is for the p2p-7-5 model. This model is ideal for symbolic bisimulation with a large number of states (2^{35}) and very few blocks after minimisation (336). For this model, our tool is 95x faster sequentially and has a parallel speedup of 8x, resulting in a total speedup of 767x. The best parallel speedup of 17x was obtained for the kanban05 model.

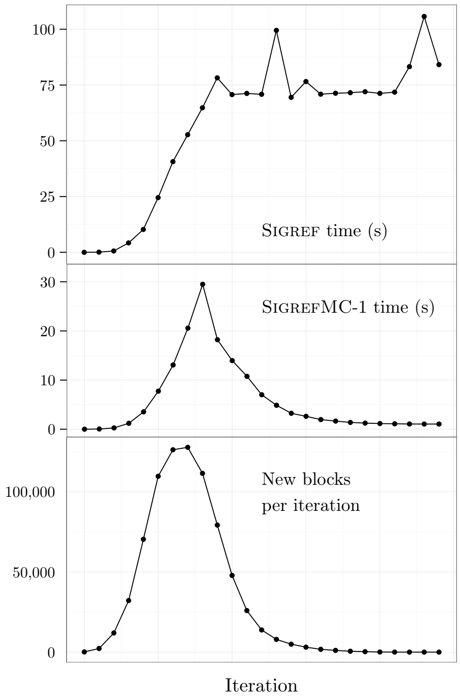

Fig. 3. Time per iteration for SIGREF and SIGREFMC (1 worker), and the number of new blocks per iteration for strong bisimulation of the kanban04 LTS model.

In almost all experiments, the signature computation dominates with 70%–99% of the execution time sequentially. We observe that the refinement step sometimes benefits more from parallelism than signature computation, with speedups up to 29.9x. We also find that reusing block numbers for stable blocks causes a major reduction in computation time towards the end of the procedure. The kanban LTS models and the larger polling CTMC models are an excellent case study to demonstrate this. See Figure 3.

6 Conclusions

Originally we intended to investigate parallelism in symbolic bisimulation minimisation. To our surprise, we obtained a much higher sequential speedup using specialized BDD operations, as demonstrated by the results in Table 1 and Figure 3. The specialized BDD operations offer a clear advantage sequentially and the integration with Sylvan results in decent parallel speedups. Our best result had a total speedup of 767x. Similar to our experiments in symbolic reachability [14], further parallel speedups might be obtained by disjunctively partitioning the transition relations.

[2] https://github.com/utwente-fmt/sigrefmc-tacas16

LTS models (strong)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
kanban03	1024240	85356	92.16	10.09	0.88	9.14	11.52	105.29
kanban04	16020316	778485	1410.66	148.15	11.37	9.52	13.03	124.06
kanban05	16772032	5033631	–	1284.86	73.57	–	17.47	–
kanban06	264515056	25293849	–	–	2584.23	–	–	–

LTS models (branching)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
kanban04	16020316	2785	8.47	0.52	0.24	16.39	2.11	34.60
kanban05	16772032	7366	34.11	1.48	0.43	22.98	3.47	79.81
kanban06	264515056	17010	118.19	3.87	0.83	30.55	4.65	142.20
kanban07	268430272	35456	387.16	8.83	1.66	43.86	5.31	232.71
kanban08	4224876912	68217	1091.67	17.91	2.98	60.96	6.02	366.72
kanban09	4293193072	123070	3186.48	34.23	5.51	93.10	6.21	578.59

CTMC models			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
cycling-4	431101	282943	220.23	26.72	2.60	8.24	10.29	84.84
cycling-5	2326666	1424914	1249.23	170.28	19.42	7.34	8.77	64.34
fgf	80616	38639	71.62	8.86	0.88	8.08	10.04	81.20
p2p-5-6	2^{30}	336	750.29	26.96	2.99	27.83	9.03	251.24
p2p-6-5	2^{30}	266	248.17	9.49	1.21	26.15	7.82	204.47
p2p-7-5	2^{35}	336	2280.76	24.01	2.97	94.99	8.08	767.12
polling-16	1572864	98304	792.82	118.50	10.18	6.69	11.64	77.85
polling-17	3342336	196608	1739.01	303.65	22.58	5.73	13.45	77.03
polling-18	7077888	393216	–	705.22	49.81	–	14.16	–
robot-020	31160	30780	28.15	3.21	0.60	8.78	5.36	47.04
robot-025	61200	60600	78.48	6.78	0.95	11.58	7.11	82.39
robot-030	106140	105270	174.30	12.26	1.47	14.21	8.33	118.44

IMC models (strong)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
ftwc01	2048	1133	1.26	1.14	0.2	1.11	5.76	6.38
ftwc02	32768	16797	154.55	102.07	15.85	1.51	6.44	9.75

IMC models (branching)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
ftwc01	2048	430	1.12	0.77	0.13	1.45	6.07	8.83
ftwc02	32786	3886	152.9	50.39	4.89	3.03	10.3	31.26

Table 1. Results for the benchmark experiments. Each data point is an average of at least 15 runs. The timeout was 3600 seconds.

References

1. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: Lightner, M.R., Jess, J.A.G. (eds.) ICCAD. pp. 188–191. IEEE Computer Society / ACM (1993)
2. Blom, S., Haverkort, B.R., Kuntz, M., van de Pol, J.: Distributed Markovian Bisimulation Reduction aimed at CSL Model Checking. ENTCS 220(2), 35–50 (2008)
3. Blom, S., Orzan, S.: Distributed Branching Bisimulation Reduction of State Spaces. ENTCS 89(1), 99–113 (2003)
4. Blumofe, R.D.: Scheduling multithreaded computations by work stealing. In: FOCS. pp. 356–368. IEEE Computer Society (1994)
5. Bouali, A., de Simone, R.: Symbolic Bisimulation Minimisation. In: Computer Aided Verification, 4th Int. Workshop. LNCS, vol. 663, pp. 96–108. Springer (1992)
6. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers C-35(8), 677–691 (aug 1986)
7. Burch, J., Clarke, E., Long, D., McMillan, K., Dill, D.: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13(4), 401–424 (apr 1994)
8. Clarke, E.M., McMillan, K.L., Zhao, X., Fujita, M., Yang, J.: Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. In: DAC. pp. 54–60 (1993)
9. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM 42(2), 458–487 (1995)
10. Derisavi, S.: A Symbolic Algorithm for Optimal Markov Chain Lumping. In: TACAS 2007. LNCS, vol. 4424, pp. 139–154. Springer (2007)
11. Derisavi, S.: Signature-based Symbolic Algorithm for Optimal Markov Chain Lumping. In: QEST 2007. pp. 141–150. IEEE Computer Society (2007)
12. van Dijk, T., Laarman, A.W., van de Pol, J.C.: Multi-Core BDD Operations for Symbolic Reachability. In: 11th Int. Workshop on Parallel and Distributed Methods in verifiCation. ENTCS, Elsevier (2012)
13. van Dijk, T., van de Pol, J.C.: Lace: Non-blocking Split Deque for Work-Stealing. In: 7th Int. Euro-Par Workshop on Multi-/Many-core Computing Systems. LNCS, vol. 8806, pp. 206–217. Springer (2014)
14. van Dijk, T., van de Pol, J.C.: Sylvan: Multi-Core Decision Diagrams. In: TACAS 2015. pp. 677–691 (2015)
15. Hermanns, H., Katoen, J.: The How and Why of Interactive Markov Chains. In: 8th Intl. Symp. Formal Methods for Components and Objects. LNCS, vol. 6286, pp. 311–337. Springer (2009)
16. Kant, G., Laarman, A., Meijer, J., van de Pol, J.C., Blom, S., van Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS 2015. pp. 692–707 (2015)
17. Kulakowski, K.: Concurrent bisimulation algorithm. CoRR abs/1311.7635 (2013)
18. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. LNCS, vol. 6806, pp. 585–591. Springer (2011)
19. Mumme, M., Ciardo, G.: An efficient fully symbolic bisimulation algorithm for non-deterministic systems. Int. J. Found. Comput. Sci. 24(2), 263–282 (2013)
20. Paige, R., Tarjan, R.E.: Three Partition Refinement Algorithms. SIAM J. Comput. 16(6), 973–989 (1987)

21. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)
22. Wijs, A.: GPU Accelerated Strong and Branching Bisimilarity Checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 368–383. Springer (2015)
23. Wimmer, R., Becker, B.: Correctness Issues of Symbolic Bisimulation Computation for Markov Chains. In: MMB&DFT. LNCS, vol. 5987, pp. 287–301. Springer (2010)
24. Wimmer, R., Derisavi, S., Hermanns, H.: Symbolic partition refinement with automatic balancing of time and space. Perform. Eval. 67(9), 816–836 (2010)
25. Wimmer, R., Herbstritt, M., Becker, B.: Optimization techniques for BDD-based bisimulation computation. In: 17th GLSVLSI. pp. 405–410. ACM (2007)
26. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref- A Symbolic Bisimulation Tool Box. In: ATVA. LNCS, vol. 4218, pp. 477–492. Springer (2006)
27. Wimmer, R., Hermanns, H., Herbstritt, M., Becker, B.: Towards Symbolic Stochastic Aggregation. Tech. rep., SFB/TR 14 AVACS (2007)