Minimization of Large State Spaces using Symbolic Branching Bisimulation

Ralf Wimmer Marc Herbstritt Bernd Becker
Institute of Computer Science,
Albert-Ludwigs-University,
79110 Freiburg im Breisgau, Germany
{wimmer,herbstri,becker}@informatik.uni-freiburg.de

Abstract: Bisimulations in general are a powerful concept to minimize large finite state systems regarding some well-defined observational behavior. In contrast to *strong* bisimulation, for *branching* bisimulation there are only tools available that work on an *explicit* state space representation. In this work we present for the first time a *symbolic* approach for branching bisimulation that uses BDDs as basic data structure and that is based on the concept of signature refinement. First experimental results for problem instances derived from process algebraic system descriptions show the feasibility and the robustness of our approach.

1 Introduction

Bisimulations are used to minimize systems where many states of the system have the same behavioral characteristics. There exist many variants of bisimulation like *weak*, *strong*, and *branching* bisimulation. Especially *branching* bisimulation, as introduced in [19, 13], is very useful in the domain of process algebras where the key aspect is the observational behavior of the compositional model description. E.g., the underlying synchronization of the modules is formally captured by introducing a so-called τ-action that allows some modules to proceed by doing *nothing* whereby one concrete other module is executing an observable action. This τ-action itself is only a vehicle to formalize such a compositional approach, but can also be used to hide non-relevant actions. Branching bisimulation can be seen as a coarsening of strong bisimulation where the branching structure of the system (e.g. respecting actions that bypass some modules) is conserved. Furthermore, today's design flow for complex systems requires the application of high-level modelling tools, e.g. Statemate [2]. There, to handle the system complexity, *symbolic* methods are applied, often by means of BDDs [6]. Therefore it is desirable to apply bisimulation algorithms already on the symbolic level, i.e., directly on the BDDs that represent the state space.
Additionally, our long term goal is to apply quantitive analysis in terms of probabilistic model checking, and this requires at the moment a reduction to explicit state space models.
Although (strong) bisimulation is often unnecessary in the context of model checking invariants (see [11]), we like to point out that we do not apply model checking during our proposed symbolic branching bisimulation. Instead we apply dedicated, non-trivial BDD-operations to the symbolic transition relation, just in order to minimize the state space. The hope is that afterwards, the state space is so small that the system can be analyzed quantitatively with current tools, e.g. [14], that work on explicit state space representations. We believe – and the experiments fortify this hypothesis – that symbolic branching bisimulation is a viable way in this context.
Furthermore, we don't want to concentrate on process algebraic system descriptions only but generally on labeled transition systems (LTSs), since this makes our approach more flexible.
In particular we can handle instances where *explicit* branching bisimulation tools like BcgMin [10] fail except for the smallest instances. We provide experimental results to show the feasibility and the robustness of our approach.

1.1 Related Work

There is a lot of work about implementing *strong* bisimulation with BDDs (e.g. [4, 9]). But only little or nothing is available about the implementation of *branching* bisimulation symbolically using BDDs. Nevertheless, we will review most related works to point out the difference of our method compared to existing bisimulation algorithms.
The main reference for branching bisimulation is the work of Groote and Vaandrager [13]. Their algorithm works on an explicit state space representation whereby iteratively so-called *splitters*, i.e., transitions that can be used for a block refinement, are computed. The Groote-Vaandrager-Algorithm is publicly available, e.g. in the CADP package [10, 12].
A first step towards symbolic computation of bisimulations was done by Bouali et al. [4], but this approach is restricted to *strong* bisimulation. Indeed, in [4] it is sketched very briefly how the approach can be extended to branching bisimulation, but no details and no experimental results are given. Additionally, the method of Bouali et al. is designed for dedicated compositional systems (i.e., it is not based on a general LTS) and this issue is also ex-

ploited for their proposed strong bisimulation algorithm. Thus, the approach of Bouali et al. seems to be focused on their specialized setting, in addition the corresponding tool is only available as a not well documented, obsolete binary executable [5]. Taken together, it is not at all clear, how to lift it to *branching* bisimulation.

Similarly, the work of [9] suggests a BDD-based approach for model checking process algebras supporting weak and strong bisimulation. Although the approach is very smart regarding to the adaption of CCS operators to BDDs, nothing is said about *branching* bisimulation.

Another approach for symbolic bisimulations in the context of stochastic process algebras was presented in [15] by Hermanns and Siegle, again focused on *strong* bisimulation. *Branching* bisimulation is not further discussed. Also, their algorithms are based on *emulating* the explicit Groote-Vaandrager algorithm.

From a preliminary analysis we gained the insight that a brute-force adoption of the Groote-Vaandrager algorithm to the symbolic setting would share some limitations of the explicit algorithm, e.g., the single splitter computation. Our hypothesis therefore is that such an adoption would be not more efficient than the explicit algorithm itself.

Fortunately, we can propose a method that works by some other means and is based on the concept of signatures as introduced by Blom and Orzan in [3]. Their algorithm for the computation of branching bisimulation of *explicit* state space representations is designed to be applied in a distributed computation environment. In this work we explain how signatures can be computed *symbolically* and why the signature refinement algorithm for computing the branching bisimulation can take advantage of the symbolic state space representation in contrast to e.g. [4].

Although the algorithm of Groote-Vaandrager is provably optimal from a computational complexity point of view in contrast to the algorithm of Blom and Orzan, that has a worst-case complexity that is one order of magnitude larger, the signature-based approach of Blom and Orzan allows an algorithm design that profits from the symbolic representation.

Taken together, this paper presents to the best of our knowledge the first algorithm exploiting the structure in a fully symbolic environment.

The outline of the paper is as follows. After reviewing preliminaries in Section 2, our main contribution, the symbolic computation of the branching bisimulation, is presented in Section 3. The evaluation of our algorithm is analyzed and discussed in Section 4. Finally, we conclude the paper in Section 5.

2 Preliminaries

In this section we will give basic definitions, notations, and algorithms to which we refer throughout the rest of the paper.
In our setting, the state spaces we are looking at are given as labeled transition systems.

Definition 1 *A* labeled transition system *(LTS) is a triple $M = (S, A, T)$ where*

- *$S \neq \emptyset$ is a finite set of states,*
- *$A \neq \emptyset$ is a finite set of actions, including the unobservable action τ,*
- *$T \subseteq S \times A \times S$ is a set of transitions, i.e.,$(s, a, t) \in T$ iff one can take a transition from state s to state t by executing action a.*

Because every bisimulation induces a partition of the state space of an LTS, we need to define the notion of a partition formally.

Definition 2 *Let S be a finite set. A set $P \subseteq 2^S$ is called a* partition *if the following conditions hold:*

$$\bigcup_{B \in P} B = S \qquad \textit{and}$$
$$\forall B_1, B_2 \in P : B_1 \neq B_2 \Rightarrow B_1 \cap B_2 = \emptyset$$

The elements of a partition P are called blocks.

A transition $(s, a, t) \in T$ is called *inert* w.r.t. a partition P, if s and t are elements of the same block of P.
For a LTS $M = (S, A, T)$ we use the following notations:

- $s \xrightarrow{a} t$ for $(s, a, t) \in T$
- $\xrightarrow{a^*}$ for the reflexive transitive closure of $\xrightarrow{a}$
- $\xrightarrow[P]{a}$ for a transition that is inert w.r.t. the partition P.
- $\xrightarrow[P]{a^*}$ for the reflexive transitive closure of $\xrightarrow[P]{a}$
- For a partition P, we denote by $P(s)$ the block of P that contains s, i.e. $P(s) = \{t \in S \mid \exists B \in P : s \in B \wedge t \in B\}$.

A branching bisimulation according to [19] can now be defined as follows.

Definition 3 *Given a LTS $M = (S, A, T)$. Then, a relation $R \subseteq S \times S$ is a* branching bisimulation *if R is symmetric and for all $s_1, s_2, t_1 \in S$ the following condition is satisfied:*
If $(s_1, t_1) \in R$ and $s_1 \xrightarrow{a} s_2$ then

$$\begin{aligned} &\textit{either} \quad a = \tau \textit{ and } (s_2, t_1) \in R \\ &\textit{or} \quad \exists t_1', t_2 \in S : t_1 \xrightarrow{\tau^*} t_1' \xrightarrow{a} t_2 \wedge \\ &\qquad (s_1, t_1') \in R \wedge (s_2, t_2) \in R \end{aligned}$$

Please note that a branching bisimulation is an equivalence relation [1].

In [3], Blom and Orzan presented a novel approach for the distributed computation of branching bisimulation for *explicit* state space representations. Their algorithm is based on analyzing the *signatures* of states w.r.t. the current partition. For completeness, we review the definition and the basic algorithm of [3]. The signature of a state can be seen as a fingerprint of the state. It collects the possible actions that can be executed in this state. Additionally, the unobservable action τ is taken into account by ignoring sequences of τ-actions that never leave the corresponding partition block of the state. Then, a refinement of a partition can be computed by dividing blocks by identifying states that have the same signature. Formally, this is captured in the following definition.

Definition 4 *Let $P = \{B_0, \ldots, B_{p-1}\}$ be a partition of the state space S. The signature* $\mathrm{sig}_P(s)$ *of a state s regarding the partition P is defined as*

$$\mathrm{sig}_P(s) = \{(a, B_i) \mid \exists s', s'' \in S : s \xrightarrow[P]{\tau^*} s' \xrightarrow{a} s'' \in B_i \wedge (a \neq \tau \vee s \notin B_i)\}.$$

The refinement $\mathrm{sigref}(P)$ *of a partition P concerning the signatures of the states is defined as*

$$\mathrm{sigref}(P) = \big\{\{s' \in S \mid \mathrm{sig}_P(s) = \mathrm{sig}_P(s')\} \bigm| s \in S\big\}.$$

As shown in [3], sigref yields a refined partition. Using this partition refinement, one can apply a fixpoint-algorithm (see Algorithm 1), for which it was proven in [3] that it computes the coarsest branching bisimulation.

Algorithm 1 Coarsest Branching Bisimulation

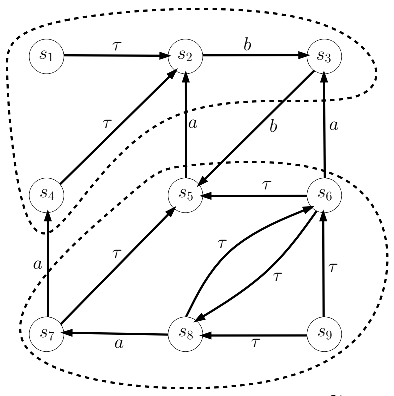
```
1: procedure BranchingBisimulation($M$)
2:     $P \leftarrow \{S\}$
3:     repeat
4:         $P' \leftarrow P$
5:         $P \leftarrow \mathrm{sigref}(P)$
6:     until $P = P'$
7: end procedure
```

As an example, let's have a look at Figure 1. We start with an LTS that is partitioned into two blocks, i.e., $P^1 = (\{s_1, s_2, s_3, s_4\}, \{s_5, s_6, s_7, s_8, s_9\}) = (p_1^1, p_2^1)$. Then, the signatures for the states w.r.t. P^1 are

$$\mathrm{sig}_{P^1}(s_1) = \mathrm{sig}_{P^1}(s_2) = \mathrm{sig}_{P^1}(s_4) = \{(b, p_1^1)\}$$
$$\mathrm{sig}_{P^1}(s_3) = \{(b, p_2^1)\}$$
$$\mathrm{sig}_{P^1}(s_5) = \mathrm{sig}_{P^1}(s_7) = \{(a, p_1^1)\}$$
$$\mathrm{sig}_{P^1}(s_6) = \mathrm{sig}_{P^1}(s_8) = \mathrm{sig}_{P^1}(s_9) = \{(a, p_1^1), (a, p_2^1)\}$$

With this, the final branching bisimulation for the example has 4 blocks:

$$P^2 = (\{s_1, s_2, s_4\}, \{s_3\}, \{s_5, s_7\}, \{s_6, s_8, s_9\}).$$



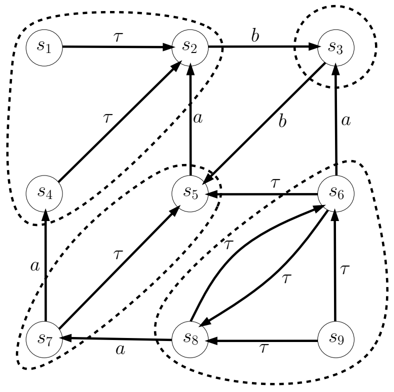


Figure 1: A LTS with an initial partition (top) and its branching bisimulation quotient (bottom).

3 Symbolic Computation

Now we will turn to symbolic representations of labeled transition systems. To compute the coarsest branching bisimulation of an LTS symbolically, we have to go into details w.r.t. the BDD representation of the LTS itself, the symbolic computation of signatures, the symbolic refinement, and finally the computation of the bisimulation quotient regarding a given partition of the state space.
We assume that the reader is familiar with BDDs and the corresponding algorithms. For a comprehensive treatment see e.g. [20, 8].

3.1 Symbolic Representation

We have to represent the following sets: the state space S, the transition relation T, the partition P, and the signatures *sig*. For the representation of the partition we assign a unique number to each block. We use a binary encoding for the states (using variables s for the current state, t for the next state and x as auxiliary variables), the actions (variables a) and the block numbers (variables k). Then, the state space is encoded by a BDD $\sigma(s)$ such that a state s is contained in the state space S if $\sigma(s) = 1$. Analogously, the transitions are represented as a BDD $\mathcal{T}(s, a, t)$ with $\mathcal{T}(s, a, t) = 1$ iff $s \xrightarrow{a} t$. Accordingly, we have a BDD $\mathcal{P}(s, k)$ with $\mathcal{P}(s, k) = 1$ iff $s \in B_k$. We use the same trick for the signatures and create a BDD $\mathcal{S}(s, a, k)$ with $\mathcal{S}(s, a, k) = 1$ iff $(a, B_k) \in \mathrm{sig}(s)$.

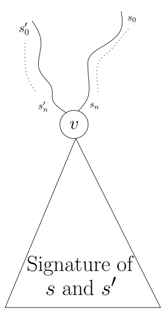


Figure 2: States that have the same signature can be identified by traversing the paths upwards starting at the signature's root node.

3.2 Computation of the Signatures

The algorithm for the symbolic computation of the signatures for the states of an LTS regarding a partition P is shown in Algorithm 2.

Algorithm 2 Computation of the Signatures

```
1: procedure SIG(transitions $\mathcal{T}$, partition $\mathcal{P}$)
2:     $reltrans(s,a,t) \leftarrow \mathcal{T}(s,a,t) \wedge (a \neq \tau \vee \neg\exists k : (\mathcal{P}(s,k) \wedge \mathcal{P}(t,k)))$
3:     $\mathcal{T}_{\tau,\mathrm{inert}}(s,t) \leftarrow \mathcal{T}(s,a,t) \wedge \neg reltrans(s,a,t)$
4:     $targets(s,a,k) \leftarrow \exists t : (reltrans(s,a,t) \wedge \mathcal{P}(t,k))$

5:     $\mathcal{C}_\tau(s,t) \leftarrow computeClosure(\mathcal{T}_{\tau,\mathrm{inert}})$
6:     $\mathcal{S}(s,a,k) \leftarrow \exists t : (\mathcal{C}_\tau(s,t) \wedge targets(t,a,k))$
7:     return $\mathcal{S}$
8: end procedure
```

In lines 2 and 3 the transition relation is split into inert τ-transitions ($\mathcal{T}_{\tau,\mathrm{inert}}$) and the remaining ones (reltrans). Thus, reltrans contains all transitions that satisfy the condition $a \neq \tau$ or connect two different blocks of the current partition.
The next step is to substitute the target state of the transition by its block number (line 4). To do so, all those states must be taken into account that can reach those states that have an outgoing relevant transition by arbitrarily many inert τ-transitions. For this, we have to compute the reflexive transitive closure of these inert τ-transitions (see for example [17] and line 5, respectively). The final step is to add the states of the closure to the signatures.

3.3 The Refinement Operation

We assume that we have already computed the BDD of the signatures of all states as described above. Now, we will compute a new partition where all states with the same signature are merged into one block.
To do so, the variable order of the BDDs must be restricted in the following way: the s_i-Variables are placed at the beginning of the variable order. This means that level(s_i) $<$ level(a_j) and level(s_i) $<$ level(k_l) hold for all i, j and l.
Now, the following observation can be exploited for the computation of the refined partition. Let s be the encoding of a state and v the BDD node that is reached when following the path from the BDD root according to s. Then the sub-BDD at v is a representation of the signature of s. All states with the same signature as s lead to the BDD-node v. To get the new block that contains s we only have to find all paths that lead to v. Then we replace the sub-BDD at v by the BDD for the encoding of the new block number k. This approach is sketched in Figure 2, and the corresponding algorithm is depicted in Algorithm 3.

Algorithm 3 Partition Refinement

```
 1: procedure REFINE(signatures $\mathcal{S}$)
 2:     if $\mathcal{S} \in$ ComputedTable then
 3:         return ComputedTable[$\mathcal{S}$]
 4:     end if
 5:     $x \leftarrow topVar(\mathcal{S})$
 6:     if $x = s_i$ then
 7:         $low \leftarrow Refine(\mathcal{S}_{\bar{x}})$; $high \leftarrow Refine(\mathcal{S}_x)$
 8:         $result \leftarrow ITE(x, high, low)$
 9:     else
10:         $result \leftarrow BDD_{cnt}(k)$; $cnt++$
11:     end if
12:     ComputedTable[$\mathcal{S}$] $\leftarrow result$
13:     return $result$
14: end procedure
```

The algorithm relies on a global variable `cnt` that is set to 0 each time we call `Refine`. It contains the number of the next block and is increased when a new block has been found. $BDD_{cnt}(k)$ is a BDD with exactly one path from the root node to the 1-leaf. The labels of the variables on this path are the binary encoding of the value of `cnt`.
Furthermore, it relies on a dynamic programming approach that stores all intermediate results in a so-called *ComputedTable*. By this it can be detected whether a node was already reached before. If a node is reached that is contained in the ComputedTable, then the corresponding entry is returned. Otherwise, if the node is not labeled with a state variable s_i, the sub-BDD at this node represents a signature that must be substituted with a new block number. If the node is labeled with a state variable, the algorithm is called recursively for the two sons of that node.

3.4 Improvements

We experienced that in most cases the BDD of the expression $\exists k : \mathcal{P}(s,k) \wedge \mathcal{P}(t,k)$, that is used for the computation of the inert τ-transitions, is considerably larger than the BDD of $\mathcal{P}(s,k)$. This can be avoided by computing the signatures and the refinement only *for one block at a time*. This means that the function SIG gets an additional parameter that contains the states of the block for which we have to compute the signatures. Then a transition is inert iff the source as well as the target state are contained in this block.
This sequential refinement of the blocks enables us to use a dedicated technique that we call *block forwarding*. After the refinement of one block, the

current partition is updated with the result of this refinement. Hence, during the refinement of the upcoming blocks this information can be used already in the same iteration and not unless the next iteration. This reduces the number of iterations needed to reach the fixpoint significantly, and results in a large runtime speedup for all considered benchmarks.

3.5 Extracting the Quotient

After applying Algorithm 1 within our BDD based framework, we have to extract the bisimulation quotient. Let $\mathcal{P}$ be a partition (given as a BDD) with sigref$(\mathcal{P}) = \mathcal{P}$. To extract the bisimulation quotient regarding this partition, we have to collapse the states of each block into one new state whose encoding is the same as the block number:

$$\sigma_{new}(s) := [k \to s](\exists s : \mathcal{P}(s,k))$$

Then, the new transition relation can be computed as follows:

$$\begin{aligned}\mathcal{R}(s,a,t) &:= [k \to t](\exists t : \mathcal{T}(s,a,t) \wedge \mathcal{P}(t,k))\\ \mathcal{T}_{new}(s,a,t) &:= [k \to s](\exists s : \mathcal{R}(s,a,t) \wedge \mathcal{P}(s,k))\end{aligned}$$

The notation $[k \to t]$ means that the k-variables must be renamed to the corresponding t-variables.

4 Experimental Results

We have implemented our approach in a tool called SIGREF using the CUDD BDD library [18]. For generating symbolic LTS representations, we have applied the stochastic process algebra tool CASPA (see [16]) from which we extracted symbolic, i.e. BDD, representations for a Kanban system [7]. The process algebraic description of the Kanban system consists of four parallel processes that are synchronized among each other. The system itself can be parameterized by a maximum number of *workpieces* that each process can handle. Since we are not aware of any available tool for *symbolic* branching bisimulation, we compared our tool SIGREF with the explicit bisimulation tool BCGMIN contained in CADP [10]. BCGMIN provides the explicit Groote-Vaandrager algorithm. Since BCGMIN requires as input an explicit description of the transition system, we had to generate a file containing all transitions explicitly (in the `.bcg` format).

We performed two series of experiments with two different configurations of the Kanban system. In the first configuration we have hidden all internal process actions, such that only the synchronization actions were visible. This kind of configuration could be of interest when inter-process communication only is analyzed. In the second configuration we have hidden all actions but the actions that are related to the first of the four processes. The motivation for this configuration is that one only wants to analyze the first process and likes to ignore the others. Our experiments were performed on an AMD Opteron Dual Processor, running with 2.6 GHz and 4 GB RAM. The results are depicted in Table 1 and 2. We have set a memory limit of 3 GB. In each case we started with an initial partition containing one block with all states.

Column 1 denotes the Kanban parameter. Column 2 gives the number of states of the original (non-minimized) transition relation including non-reachable states, column 3 denotes the number of states of the corresponding bisimulation quotient (i.e. the minimized system). The running times for SIGREF and BCGMIN are contained in column 4 and 7, respectively. Since the limiting factor for the explicit algorithm is the memory requirement, column 8 denotes the memory peak of BCGMIN. Please note that we have set a memory limit of 3 GB.[1] In column 6 the maximal number of BDD nodes created during the refinement process is displayed.

The memory requirements of SIGREF grow very moderately by an average factor of 1.69 (1.66) for the first (second) configuration whereby the state spaces grows by an average factor of 10.80 (10.80). The mean factor for the running times of SIGREF is 7.95 (9.31) for increasing Kanban parameter. This suggests that the runtime is more closely related to the state space size than to the memory requirement, which is not astonishing since the algorithm must analyze every state of the state space in some way to compute the refinement.

Nevertheless, the symbolic approach computes the bisimulation for all but the largest two Kanban instances in less than 90 minutes. But given enough time SIGREF is even able to minimize systems with more than 4 billion states – a number far out of reach for current *explicit* bisimulation tools. For the smaller instances that both tools are able to minimize, the runtimes of SIGREF and BCGMIN are comparable, whereby SIGREF is mostly faster.

In Table 3 we have compared SIGREF regarding our block forwarding technique. The results make clear that block forwarding often shortens the number of iterations immense. Hence, the overall runtimes are drastically improved.

5 Conclusions

In this work we proposed a fully symbolic approach for the computation of *branching* bisimulation. Our results show that we are able to overcome limitations of the explicit algorithm.

One main target for future work is to integrate our approach into a fault-tree based analysis tool for system dependability. For this we will conduct further research in direction of extending the signature-based approach to other bisimulations, e.g. *safety* bisimulation, and hopefully also to stochastic variants thereof.

[1]A column entry 'memout' means that the memory limit was exceeded.

p	# states original	# states bisim.	time SIGREF	memory SIGREF	node peak SIGREF	time BCGMIN	memory BCGMIN
1	256	24	0.02s	9 MB	16352	0.19s	3 MB
2	63808	206	0.58s	23 MB	435372	0.47s	8 MB
3	1024384	872	11.20s	59 MB	2212630	8.73s	92 MB
4	16021157	2785	1m 38s	63 MB	2313808	2m 46s	1457 MB
5	16772096	7366	9m 15s	67 MB	2483460	7m 48s	2365 MB
6	264515056	17010	40m 24s	76 MB	2704212	—	memout
7	268430272	35456	4h 49m 36s	124 MB	5037438	—	memout
8	4224876912	68217	11h 59m 00s	244 MB	11281858	—	memout

Table 1: Results for the Kanban system where process internal actions are hidden.

p	# states original	# states bisim.	time SIGREF	memory SIGREF	node peak SIGREF	time BCGMIN	memory BCGMIN
1	256	32	0.02s	9 MB	19418	0.17s	3 MB
2	63808	200	0.40s	15 MB	308644	0.53s	8 MB
3	1024384	800	4.54s	54 MB	1897854	10.46s	93 MB
4	16021157	2540	1m 13s	62 MB	2249422	3m 42s	1474 MB
5	16772096	6272	9m 49s	66 MB	2392502	13m 04s	2400 MB
6	264516992	14112	1h 28m 57s	74 MB	2561132	—	memout
7	268430336	28800	9h 10m 15s	117 MB	4752300	—	memout
8	4224885193	54450	46h 59m 33s	181 MB	7823410	—	memout

Table 2: Results for the Kanban system where all but the actions related to the first process are hidden.

p	# iterations with	# iterations without	time [sec] with	time [sec] without
1	4	5	0.02s	0.03s
2	6	8	0.58s	0.66s
3	7	11	11.20s	19.09s
4	8	14	1m 38s	3m 01s
5	8	17	9m 15s	17m 39s
6	8	20	40m 24s	2h 09m 41s
7	10	23	4h 49m 36s	13h 25m 44s

Table 3: Comparison of SIGREF with and without block forwarding for the same benchmarks as in Table 1

References

[1] T. Basten. Branching bisimilarity is an equivalence indeed! In *Information Processing Letters*, volume 58(3), pages 141–147, 1996.

[2] T. Bienmüller, W. Damm, and H. Wittke. The STATEMATE Verification Environment - Making It Real. In *Proc. of CAV*, pages 561–567, 2000.

[3] S. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. In *Proc. of 2nd International Workshop on Parallel and Distributed Model Checking*, 2003.

[4] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Proc. of CAV*, volume 443 of *LNCS*, pages 96–108, 1992.

[5] A. Bouali, A. Ressouche, R. de Simone, and V. Roy. *The FC-tools User Manual.*

[6] R. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[7] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. Technical Report 96-35, NASA Research Center, 1996.

[8] R. Drechsler and B. Becker. *Binary Decision Diagrams – Theory and Implementation.* Kluwer Academic Publishers, 1998.

[9] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.

[10] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In *Proc. of CAV*, pages 437–440, 1996.

[11] K. Fisler and M. Y. Vardi. Bisimulation and model checking. In L. Pierre and T. Kropf, editors, *CHARME*, volume 1703 of *LNCS*, pages 338–341. Springer, 1999.

[12] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, 2002.

[13] J. F. Groote and F. W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M. S. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *LNCS*, pages 626–638. Springer, 1990.

[14] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A markov chain model checker. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2000.

[15] H. Hermanns and M. Siegle. Bisimulation Algorithms for Stochastic Process Algebra and Their BDD-Based Implementations. In J.-P. Katoen, editor, *ARTS*, volume 1601 of *LNCS*, pages 244–264. Springer, 1999.

[16] M. Kuntz, M. Siegle, and E. Werner. Symbolic performance and dependability evaluation with the tool CASPA. In M. Núñez, Z. Maamar, F. L. Pelayo, K. Pousttchi, and F. Rubio, editors, *FORTE Workshops*, volume 3236 of *LNCS*, pages 293–307. Springer, 2004.

[17] Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transition relation. In *Proc. of DAC*, pages 260–265, 1993.

[18] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.4.1.* University of Colorado at Boulder, 2005.

[19] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[20] I. Wegener. *Branching programs and binary decision diagrams.* SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000. Theory and applications.