Optimization Techniques for BDD-based Bisimulation Computation*

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

ABSTRACT

In this paper we report on optimizations for a BDD-based algorithm for the computation of bisimulations. The underlying algorithmic principle is an iterative refinement of a partition of the state space. The proposed optimizations demonstrate that both, taking into account the algorithmic structure of the problem and the exploitation of the BDD-based representation, are essential to finally obtain an efficient symbolic algorithm for real-world problems.

The contributions of this paper are (1) *block forwarding* to update block refinement as soon as possible, (2) *split-driven refinement* that over-approximates the set of blocks that must definitely be refined, and (3) *block ordering* to fix the order of the blocks for the refinement in a clever way.

We provide substantial experimental results on examples from different applications and compare them to alternative approaches when possible. The experiments clearly show that the proposed optimization techniques result in a significant performance speed-up compared to the basic algorithm as well as to alternative approaches.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*

Keywords

Symbolic methods, binary decision diagrams, state space explosion, bisimulation, state space reduction

General Terms

Algorithms, Verification

1. INTRODUCTION

*This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'07, March 11–13, 2007, Stresa-Lago Maggiore, Italy.
Copyright 2007 ACM 978-1-59593-605-9/07/0003 ...$5.00.

This paper reports on successful optimization techniques for BDD-based computation of bisimulations. In the context of labeled transition systems bisimulations are equivalence relations in terms of observational behavior. The resulting bisimulation quotient is a compressed representation of the original state space that preserves the observational behavior. The relationship of bisimulations to formal verification techniques (e.g., Model Checking) is that bisimulations preserve important logical properties, e.g., CTL*\X[1] in case of branching bisimulation [20]. Although symbolic bisimulation computation seems not to pay off in the context of *symbolic* invariant checking [11], it plays an essential role when sophisticated analysis techniques are applied that require an explicit state space representation (see e. g. [3]). Moreover, bisimulations are an elementary method that is widely spread in verification [22]. They are often the means of choice to circumvent the inevitable state space explosion problem without including problem specific knowledge.

In our case, we have already successfully coupled a basic version of a BDD-based bisimulation computation with probabilistic model checking tools [3] that are still restricted to "small" state spaces up to 10^8 states [16], and de facto rely on an explicit state space representation. Contrary, high-level specification methodologies, e. g., STATEMATE [13], lead in practice to huge state spaces that are far out of reach for an explicit representation. Consequently, BDD-based model representations are used and BDD-based bisimulation is a way to reduce such state spaces. Furthermore, system models often contain a lot of redundancies which in fact are exactly the reason why bisimulation techniques are applicable, i. e., the state space is compressible. These redundancies are stemming either from component reuse (e. g., within a hierarchical or concurrent design approach) or from symmetries inherent to the model (e. g., schedulers).

In previous works, we have shown the general feasibility of the BDD-based computation of branching bisimulation using the concept of *signatures* [24]. Additionally, we extended our approach to the most important types of bisimulation known from the literature (e. g. weak, orthogonal bisimulation) [25], thus enabling a flexible handling of different bisimulations. In [3] we performed a case study to quantitatively analyze large STATEMATE models.

The main contribution of this work are novel optimization techniques that on the one hand take into account the algorithmic structure of the signature-based refinement algorithm and on the other hand exploit the effectiveness of

[1] CTL*\X is the computation tree logic where the next-state quantifier 'X' is not allowed. Furthermore, for an application of pure CTL see e.g. [5].

the BDD-based problem representation for efficiently computable, but yet powerful, heuristics for determinization of the refinement ordering. Only these optimizations make the application to practical models possible.

The experimental evaluation of our approach is done for examples stemming from quite different domains and clearly show that the proposed optimization techniques drastically improve our basic algorithm. Additionally, we present experimental comparisons to other symbolic methodologies, which emphasize that our approach is much more robust.

The paper is structured as follows. In the next section we will give preliminaries that are used throughout the paper. In Section 3 we will then briefly describe the BDD-based bisimulation framework whereby we focus on branching bisimulation. Our novel optimization techniques are described in detail in Section 4. The experimental evaluation of our proposed techniques and a discussion of the results follow in Section 5. Section 6 concludes the paper and states some ideas for future work.

2. PRELIMINARIES

Bisimulations typically define equivalent behavior of states in a discrete state space. There are mainly two different formalisms: state-labeled systems (e. g. Kripke-structures) and transition-labeled systems. In this work we will focus on the latter.

Definition 1 *A* labeled transition system *(LTS) is a triple $M = (S, A, T)$ where S is a finite non-empty set of states, A is a set of actions that may contain the so-called non-observable action τ, and $T \subseteq S \times A \times S$ is a relation that defines labeled transitions between states.*

A bisimulation partitions the state space S into disjoint parts.

Definition 2 *Given a finite set S, a* partition *of S is a set $P \subseteq 2^S$ with*

$$\bigcup_{B \in P} B = S \quad \text{and} \quad \forall B, B' \in P : B = B' \vee B \cap B' = \emptyset$$

The elements B of the partition are called blocks.

Definition 3 *Let P and P' be two partitions of the same set S. Then P is a* refinement *of P' (or conversely P'* coarser *than P) if $\forall B \in P \, \exists B' \in P' : B \subseteq B'$.*

For a transition system $M = (S, A, T)$ and a partition P of S we use the following notations:

- $(s, t) \in P$ if there is a $B \in P$ with $s \in B$ and $t \in B$.
- $s \xrightarrow{a} t$ for $(s, a, t) \in T$
- $s \xrightarrow{a^*} t$ for the reflexive transitive closure of $\xrightarrow{a}$
- $s \xrightarrow[P]{a^*} t$ if $s \xrightarrow{a^*} t$ and $(s, t) \in P$. We call such a sequence *inert*.

Bisimulations are equivalence relations on the state space of an LTS. They define which states are considered indistinguishable. There is a large number of different bisimulation one can choose from, depending on the application [22]. In this paper, we will concentrate on one of the most important bisimulation, namely on branching bisimulation. All optimizations that will be presented in this section apply in the same way to other types of bisimulation that are described in [25].

Branching bisimulation was introduced by van Glabbeek and Weijland [23] to overcome the problem of weak bisimulation, which does not preserve the branching structure of the LTS. Branching bisimulation is comparable to stuttering equivalence on Kripke structures and preserves CTL* without next-state quantifier [20].

Definition 4 *A binary relation $\mathfrak{B}_b \subseteq S \times S$ is a* branching bisimulation[2] *if for all $s, s', t \in S$ the following holds: If $(s, t) \in \mathfrak{B}_b$ then $s \xrightarrow{a} s'$ implies either $a = \tau$ and $(s', t) \in \mathfrak{B}_b$ or there exist t', t'', $t''' \in S$ with $t \xrightarrow[\mathfrak{B}_b]{\tau^*} t' \xrightarrow{a} t'' \xrightarrow[\mathfrak{B}_b]{\tau^*} t'''$ and $(s', t''') \in \mathfrak{B}_b$.*

The fastest known *explicit* algorithm for computing the coarsest branching bisimulation of a transition system is that of Groote and Vaandrager [12]. Short descriptions how symbolic algorithms for strong bisimulation can be modified for branching bisimulation can be found in [4, 10]. Blom and Orzan have shown in [2] how a signature-based approach can be used for the distributed minimization of explicitly represented state spaces using branching bisimulation. In the next section, we will briefly review our previous work on extending the signature-based approach to BDD-based state space representations.

3. SYMBOLIC COMPUTATION

Before describing the symbolic part of the bisimulation computation, we will first explain the principles of the signatures and the refinement operation. The next step is then to explain how the required data is represented by BDDs and how the signatures and the iterative refinement can be computed symbolically.

3.1 Signature-based Partition Refinement

The signature-based approach as first introduced by Blom and Orzan in [2] relies on the notion of the signature of a state. A signature can be considered as a kind of "fingerprint" that characterizes the state w. r. t. the observable actions that can be executed after an arbitrary number of unobservable τ-steps.

Definition 5 *The* branching signature *of a state $s \in S$ w. r. t. a partition P is defined as*

$$\begin{aligned}\mathrm{sig}_P(s) = \big\{ (a, B) \in A \times P \mid \\ \exists s' \in S, s'' \in B : s \xrightarrow{\tau^*} s' \xrightarrow{a} s'' \\ \wedge (s, s') \in P \wedge (a \neq \tau \vee (s, s'') \notin P) \big\}.\end{aligned}$$

The refined partition is then obtained by splitting the blocks according to the different signatures of its states.

Definition 6 *For a partition P and a block B of P, the* refinement operator sigref *is given by*

$$\mathrm{sigref}_P(B) = \big\{ \{ t \in B \mid \mathrm{sig}_P(t) = \mathrm{sig}_P(s) \} \mid s \in B \big\}$$
$$\mathrm{sigref}(P) = \bigcup_{B \in P} \mathrm{sigref}_P(B).$$

[2]Please note that the definition in [25, 24] is slightly different. But according to the stuttering lemma from [23], they are equivalent.

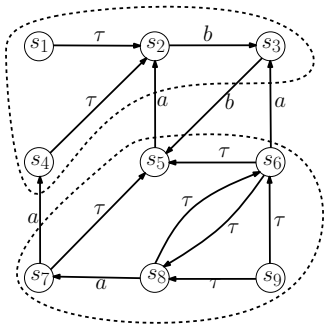

Figure 1: An example of an LTS with initial partition.

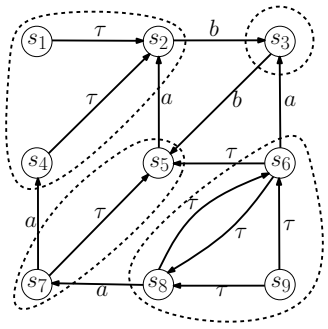

Figure 2: The coarsest branching bisimulation of the LTS in Figure 1 refining the given initial partition.

Blom and Orzan have shown in [2] that this yields the coarsest branching bisimulation if iterated until a fixpoint is reached.

We will now illustrate this concept with a small example:

Example 1 *As an example we use the LTS of Figure 1 together with the given initial partition P^0 with two blocks, i.e., $P^0 = (\{s_1, s_2, s_3, s_4\}, \{s_5, s_6, s_7, s_8, s_9\}) = (B_1^1, B_2^1)$. Then, the signatures for the states w. r. t. P^0 are*

$$\begin{aligned} \mathrm{sig}_{P^0}(s_1) = \mathrm{sig}_{P^0}(s_2) = \mathrm{sig}_{P^0}(s_4) &= \{(b, B_1^1)\} \\ \mathrm{sig}_{P^0}(s_3) &= \{(b, B_2^1)\} \\ \mathrm{sig}_{P^0}(s_5) = \mathrm{sig}_{P^0}(s_7) &= \{(a, B_1^1)\} \\ \mathrm{sig}_{P^0}(s_6) = \mathrm{sig}_{P^0}(s_8) = \mathrm{sig}_{P^0}(s_9) &= \{(a, B_1^1), (a, B_2^1)\} \end{aligned}$$

With this, the refined partition (which is also the final branching bisimulation) for the example has four blocks:

$$P^1 = (\{s_1, s_2, s_4\}, \{s_3\}, \{s_5, s_7\}, \{s_6, s_8, s_9\}).$$

3.2 Representation of the Data

For the symbolic bisimulation computation we have to represent the following sets: the state space S of the LTS, its transition relation T, the partition P and the signatures sig.

We use a binary encoding for the states (using variables s for the present state, variables t for the next state, and x as auxiliary variables) and the actions (variables a). Then, the state space is represented by a BDD $\mathcal{S}$ with $\mathcal{S}(s) = 1$ iff $s \in S$. Analogously, we have a BDD $\mathcal{T}$ for the transition relation with $\mathcal{T}(s, a, t) = 1$ iff $s \xrightarrow{a} t$.

We have chosen an uncommon way for the representation of the partition P: We assign a unique number to each block of P (encoded using variables k) and represent P by a BDD $\mathcal{P}$ with $\mathcal{P}(s, k) = 1$ iff $s \in B_k$.

Algorithm 1 Signature for Branching Bisimulation

```
1: procedure SigBranching(LTS M, partition P)
2:     T_τ(s,t) ← Cofactor(T(s,a,t), a = τ)
3:     inert_τ(s,t) ← Closure(T_τ(s,t)) ∧ ∃k : (P(s,k) ∧ P(t,k))
4:     pre(s,a,t) ← ∃x : (inert_τ(s,x) ∧ T(x,a,t))
                         ∧ (a ≠ τ ∨ ¬∃k : P(s,k) ∧ P(t,k))
5:     return ∃t : pre(s,a,t) ∧ P(t,k)
6: end procedure
```

All other symbolic algorithms for bisimulations (e. g. [4]) typically use a BDD $\mathcal{P}'(s, t)$ with $\mathcal{P}'(s, t) = 1$ iff $(s, t) \in P$. Our representation has two advantages: First, our experiments have shown that mostly $\mathcal{P}'$ is much larger than $\mathcal{P}$ in terms of BDD nodes. Second, given $\mathcal{T}$ and $\mathcal{P}$, it is easy to compute the quotient w. r. t. P symbolically (see [25]). We represent the signatures in the same way and create a BDD σ with $\sigma(s, a, k) = 1$ iff $(a, B_k) \in \mathrm{sig}(s)$.

3.3 Computation of the Signatures

For the computation of the branching signature we provide several basic BDD operations:

- Extraction of the τ-transitions from $\mathcal{T}$:
  $\mathit{Cofactor}(\mathcal{T}(s, a, k), a = \tau)$
- Pairs of states (s, t) that are in the same block:
  $\exists k : \mathcal{P}(s, k) \wedge \mathcal{P}(t, k)$
- Computation of the reflexive transitive closure (RTC) of a relation $R(s, t)$: There are several symbolic algorithms for the computation of the RTC (e. g. [7, 18]). We apply the iterative squaring method of [7].
- Computation of the non-τ or non-inert transitions:
  $\mathcal{T}(s, a, t) \wedge \neg\big(\exists k : (P(s, k) \wedge P(t, k)) \wedge a = \tau\big)$
- Concatenation of $R_1(s, t)$ and $R_2(s, t)$:
  $\exists x : R_1(s, x) \wedge R_2(x, t)$
- Substitution of t in $R(s, t)$ by its block number:
  $\exists t : R(s, t) \wedge P(t, k)$

Algorithm 1 shows how these basic operations can be combined to compute the branching signature.

At first, all pairs of states that are connected by a τ-transition and their RTC, such that source and target state are contained in the same block, are computed. In line 4 the closure of the τ-transitions and the observable transitions are concatenated. Finally, in line 5 the target state of the transition sequence is replaced by its block number.

3.4 Computation of the Refinement

We assume that we have already computed the BDD $\sigma(s, a, k)$ for the signatures as described above. We are now going to show how to compute the refined partition such that all states with the same signature are merged into one block.

The variable order of the BDD is constrained as follows: The s_i variables are placed at the top of the variable order, followed by the a_j and k_l variables, i. e., $\mathrm{level}(s_i) < \mathrm{level}(a_j)$ and $\mathrm{level}(s_i) < \mathrm{level}(k_l)$ must hold for all i, j, and l.

Then, we can exploit the following observation: Let s be the encoding of a state. If we follow the path given by s in the BDD, we reach a node v. The sub-BDD at node v represents the signature of s. Furthermore, all states with the same signature as s lead to v, due to the canonicity of BDDs. To get the refined partition, we have to replace all sub-BDDs that represent the signature of a state $s \in S$ by

Algorithm 2 Partition Refinement

```
1: procedure REFINE(signatures σ)
2:     if σ ∈ ComputedTable then
3:         return ComputedTable[σ]
4:     end if
5:     x ← topVar(σ)
6:     if x = s_i then
7:         low ← Refine(Cofactor(σ, x = 0))
8:         high ← Refine(Cofactor(σ, x = 1))
9:         result ← BDDnode(x, high, low)
10:    else
11:        result ← newBlockNumber()
12:    end if
13:    ComputedTable[σ] ← result
14:    return result
15: end procedure
```

the BDD for the encoding of a new block number k. This is sketched in Algorithm 2.

The algorithm uses a function `newBlockNumber()` that returns a BDD with exactly one path from the root node to the leaf 1. The values of the variables on that path correspond to the binary encoding of a block number that has not been used in the current partition BDD. It is reset each time we call `Refine`.

Furthermore, the algorithm relies on a dynamic programming approach to store all intermediate results in a so-called *ComputedTable*. By this, we can detect whether a node was reached before. If we reach a node already contained in the *ComputedTable*, then we return the stored result. Otherwise, if the node is labeled with a state variable s_i, the algorithm is called recursively for the two sons. If the label of the node is not a state variable, then the node is the root of a sub-BDD representing a signature. This node must be substituted with a new block number. For this, the ComputedTable has to be perfect (and is not a cache as usual).

The complexity of the refinement is linear in the size of $\sigma(s, a, t)$ if we assume that accessing the *ComputedTable* and the call to `newBlockNumber()` take constant time.

4. OPTIMIZATIONS

In this section we present several optimizations that increase the efficiency of the basic algorithm described in the previous section.

4.1 Block Forwarding

During our experiments we observed that the BDD for the expression

$$\exists k : P(s, k) \wedge P(t, k), \tag{1}$$

that describes the pairs of states that are contained in the same block (see lines 3 and 4 of algorithm 1), is considerably larger than the BDD for $P(s, k)$, and the variable order with the s_i and t_i variables at the beginning and the k_i variables at the end makes the computation very expensive.

However, the computation of (1) can be avoided if the signature computation and the refinement are carried out not for all blocks in one step, but only for one block at a time. Hence, we modified the function `SigBranching` such that it takes an additional parameter $\mathcal{B}(s)$ for the block we have to compute the signatures for. Then, we can replace expression (1) by $\mathcal{B}(s) \wedge \mathcal{B}(t)$. Experiments have shown that block-by-block refinement pays off for all our benchmarks.

The block-wise refinement enables us to apply the following simple optimization technique, which we call *block forwarding*: After the refinement of a block, the current partition is updated with the result of this refinement. Hence, during the refinement of the remaining blocks this information can already be used in the same iteration. Block forwarding substantially reduces the number of iterations needed to reach the fixpoint.

Algorithm 3 Signature-based Refinement

```
1: procedure BISIMULATION
2:     P ← initial partition                    ▷ current partition
3:     U ← P                                    ▷ unstable blocks
4:     U_new ← ∅          ▷ unstable blocks for the next iteration
5:     while U ≠ ∅ do
6:         for all blocks B ∈ U do
7:             P ← (P \ {B}) ∪ sigref_P(B)
8:             U_new ← U_new \ {B}
9:             if B was split then
10:                U_new ← U_new ∪ bw_sig^oa_P(B)
11:            end if
12:        end for
13:        U ← U_new
14:    end while
15:    return P
16: end procedure
```

4.2 Split-driven Refinement (SDR)

Especially at the end of the refinement process only few blocks are split while the majority of the blocks remains unchanged. Therefore we would like to ignore blocks of which we know a priori that they will not be split. To do so, assume that in the current iteration the block B was split. Then, only those blocks are potentially unstable that contain a state that has a pair (a, B) in its signature, for some action a. To capture these potentially unstable states, we define the backward signature $\mathrm{bw\_sig}_P(B)$ as follows:

$$\mathrm{bw\_sig}_P(B) = \{B' \in P \mid \exists s \in B' \, \exists a \in A : (a, B) \in \mathrm{sig}_P(s)\}$$

That means, $\mathrm{bw\_sig}_P(B)$ is exactly the set of blocks that may be affected by the splitting of B. The signatures of all states contained in other blocks have not changed. Now, if we computed $\mathrm{bw\_sig}_P$ in that way, we would encounter the same problem as during the computation of the signatures: For extracting the inert τ-transitions we have to compute the BDD for expression (1). But in this case we cannot avoid it easily by block-wise refinement because we walk backwards in the LTS. Fortunately, we are not forced to compute $\mathrm{bw\_sig}_P(B)$ properly, but it suffices to compute an *over-approximation* $\mathrm{bw\_sig}^{oa}_P$ of the backward signature, e. g. by saying that τ-steps are relevant for the backward signature even if they are inert.

$$\mathrm{bw\_sig}^{oa}_P(B) = \{B' \in P \mid \exists s' \in B' \, \exists a \in A \, \exists s \in B : s' \xrightarrow{a} s\}$$

This may cause some unnecessary refinements of stable blocks, but it does not impact the correctness of the result.

Algorithm 3 shows the pseudo-code of the refinement algorithm with all of the presented optimization techniques. This version will be used for our experiments in section 5.

4.3 Choosing a Good Order of the Blocks

The effectiveness of the block forwarding technique strongly depends on the order in which the potentially unstable blocks are refined (see line 6 of algorithm 3). The idea is to provide a block order such that blocks having a great impact on the stability of others are split first. Then, more blocks may be split in one iteration than without any ordering. Before the actual refinement, we therefore sort the potentially unstable blocks w. r. t. one of the following block ordering heuristics:

• *SortByBWSig*: in decreasing order w. r. t. $|\bigcup \text{bw\_sig}^{oa}(B)|$
• *SortByBlockSize*: in decreasing order w. r. t. $|B|$.

Please note that both heuristics can be computed efficiently using the BDD-function *satisfy_count* [6] that is linear in the size of the BDD.

We will show in Section 5 that these orders often have the intended effect, namely that they reduce the number of potentially unstable blocks that have to be refined.

5. EXPERIMENTAL RESULTS

To evaluate our algorithm we implemented it in a tool called SIGREF [25] using the BDD package CUDD [21]. For comparison with other state-of-the-art algorithms, we also implemented an extension of Bouali/de Simone's algorithm [4] to branching bisimulation, as briefly suggested in their paper. We applied both tools to three different sets of benchmarks stemming from very different domains:

Kanban Production System Here, we use a process-algebraic description of a Kanban system [8] that models a production environment with four machines each having a parameterizable buffer of workpieces. From this description we generated a BDD representation of the transition system using the CASPA tool [17]. CASPA allows action-hiding, and therefore, as an example, we have hidden all internal actions that are not involved in the synchronization of the machines. This is the appropriate configuration when only inter-process communication will be analyzed.

Milner's Scheduler The second group of benchmarks is an implementation of Milner's scheduler [19]. Its purpose is to schedule n not further specified tasks $T_1, \ldots, T_n$ with the following constraints: T_i must be started before T_{i+1}. After T_n, T_1 is started again. T_i may only be started again if its previous run has already terminated. To enforce the ordering constraint we use a token-passing mechanism. A task may only be started if it holds the token and is not running. The token is then passed to the next task. For the analysis, we assume a scenario in which we want to prove that the ordering constraint is satisfied. Therefore we map all other actions (e. g. those for the token passing or the termination of the tasks) to τ.

Failure-enhanced Statemate descriptions As a third benchmark suite we analyzed LTSs that were generated from STATEMATE descriptions [13] that are extended by some failure-behavior. The first example describes a train control system stemming from the ETCS specification and models a scenario regarding the communication between trains and the Radio Block Centers (RBCs) (see [9] for details about ETCS which is part of ERTMS). The analysis tackles the problem of colliding trains on the same track. The example is scalable in the number of trains whereby we used 1, 2, and 3 trains, resulting in three benchmarks *etcs-1*, *etcs-2*, and *etcs-3*. Especially *etcs-3* samples a realistic scenario. Furthermore, we used an example, *bs-p*, from the ARP 4761 case study [1] that models a braking system from an air-plane. It is about the correctness of the pilot's braking pedal and the hydraulic pressure given to the wheels of the air-plane. The benchmark *ctrl* describes a redundancy controller of an industrial avionics project. A detailed description of all STATEMATE models can be found in [14].

Table 1 shows the sizes of all benchmarks before and after minimization. We applied four different versions of SIGREF: (1) without SDR and no block ordering, (2) with SDR and no block ordering, (3) with SDR and *SortByBlockSize* block ordering, and (4) with SDR and *SortByBWSig* block ordering. Additionally, we applied Bouali/de Simone's algorithm to the benchmarks. The results[3] are contained in Table 2. The number of iterations needed to reach the fixpoint is denoted by "#it", the columns denoted by "#refined" contain the number of blocks that were refined during the computation. It can be seen that the application of split-driven refinement substantially reduces this number by a factor of at least 2. This effect is intensified by the application of an appropriate ordering heuristics – especially for the STATEMATE benchmarks. The reduction of the number of refined blocks is also resembled in the runtimes: the smaller the number of refined blocks, the smaller the runtime. The few exceptions are caused by the higher computational costs of SortByBWSig. On the average, the SortByBlockSize heuristics gives the best results w. r. t. CPU time.

Benchmark	state space before	state space after	transitions before	transitions after
kanban-4	16020316	2785	74424320	10932
kanban-5	16772032	7366	133938560	31795
kanban-6	264515056	17010	1689124864	78584
kanban-7	268430272	35456	2617982976	172382
kanban-8	4224876912	68217	29070458880	345128
kanban-9	4293193072	123070	41055336960	642837
milner-4	4096	252	20480	1020
milner-5	32768	1019	204800	5115
milner-6	262144	4090	1966080	24570
milner-7	2097152	16377	18350080	114681
milner-8	16777216	65528	167772160	524280
etcs-1	1057	51	15058	749
etcs-2	428113	1312	16589262	48848
etcs-3	158723041	35842	16658393318	3128876
bs-p	184865921	1177	10025344274	42830
ctrl	139623	9627	11867888	653303

Table 1: Size of the benchmarks

We are aware of the runtimes of Bouali/de Simone's algorithm for the examples of Milner's scheduler that outperform SIGREF by orders of magnitude. This is mainly due to the very regular structure of the scheduler that generates BDDs for $\mathcal{P}'(s,t)$ with only a few hundred nodes.

The opposite happens for the more irregular benchmarks, mainly those generated from STATEMATE designs, which are — from our point of view — much more important in practice: SIGREF manages to compute the bisimulation quotient in reasonable time whereas Bouali/de Simone's algorithm fails due to the time limit. The reason for this is again the size of the partition BDD: because of the more irregular structure the size of $P'(s,t)$, which is used by Bouali/de Simone, is much bigger than our representation.

As a summary we can say that although there are cases in which other tools outperform SIGREF due to special structures of the state space, SIGREF is much more robust and can cope with a wide range of benchmarks. Furthermore, the proposed optimizations are able to reduce the runtime significantly compared to the basic algorithm.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented powerful optimization techniques for a signature-based symbolic bisimulation framework that allow the minimization not only of larger state spaces, but also in less time.

We have provided experimental results for benchmarks stemming from different domains such as STATEMATE mod-

[3]The experiments were performed on an AMD Opteron 2.6 GHz CPU. We have set a time limit of 8 h (= 28 800 s). An entry "TL" means that this time limit was exceeded. In all cases, the tools needed less than 512 MB of memory.

	Sigref without SDR			Sigref with split-driven refinement									Bouali	
				No ordering			SortByBlockSize			SortByBWSig				
Benchmark	#it.	#refined	Time [s]	#it.	#refined	Time [s]	#it.	#refined	Time [s]	#it.	#refined	Time [s]	#it.	Time [s]
kanban-4	8	13879	35.89	10	4747	12.65	9	4438	13.04	11	4504	15.28	14	5.20
kanban-5	8	34563	178.63	8	11813	56.62	10	11310	61.16	13	11266	72.34	17	23.17
kanban-6	8	74056	753.32	8	27163	252.66	12	25609	346.06	10	26364	314.01	20	68.73
kanban-7	10	213787	4163.36	10	56406	943.51	14	53273	1061.32	11	52116	1361.09	23	621.42
kanban-8	10	388169	17383.70	10	108270	3239.49	12	101907	3330.95	12	101678	3743.23	n. a.	TL
kanban-9	n. a.	n. a.	TL	12	200414	10098.50	14	174999	9736.97	13	178144	11034.00	n. a.	TL
milner-4	5	747	0.15	7	320	0.11	6	387	0.14	6	370	0.15	7	0.02
milner-5	5	2970	2.40	9	1285	1.12	8	1389	1.28	8	1421	1.41	9	0.05
milner-6	5	11833	29.06	11	5122	12.40	8	5380	12.99	8	5346	13.80	10	0.12
milner-7	5	47224	384.15	13	20415	161.29	12	20563	155.75	10	20973	167.90	12	0.24
milner-8	5	188663	6376.51	10	81468	2250.05	12	78829	2215.16	11	79591	2219.40	13	0.49
etcs-1	6	171	0.11	6	88	0.08	6	76	0.09	6	77	0.10	5	2.29
etcs-2	8	6179	49.77	8	2246	20.73	8	1789	20.62	8	1749	21.56	n. a.	TL
etcs-3	9	203986	17944.40	9	63622	5899.97	9	50540	4249.97	9	47465	4265.11	n. a.	TL
bs-p	14	10981	189.42	14	6976	150.05	13	3792	98.70	14	4011	107.43	n. a.	TL
ctrl	10	55964	1981.02	10	27254	818.48	10	19653	576.12	8	21227	612.71	n. a.	TL

Table 2: Benchmark results for different Sigref versions

els and process-algebraic descriptions that show the efficacy of our improvements. In comparison with other symbolic algorithms, the results show that our approach is much more robust regarding a wide range of transition systems.

As future work, we will investigate how the signature-based approach can be extended to compute *stochastic* bisimulations for Interactive Markov Chains [15]. Furthermore, we will think about a flexible way to combine both the signature-based approach as well as Bouali/de Simone's algorithm.

Acknowledgements
We would like to thank the whole AVACS::S3 team for its fruitful cooperation. Especially, we thank Thomas Peikenkamp and Eckard Böde for providing the STATEMATE examples. Additionally, we are deeply grateful to Markus Siegle and Matthias Kuntz for the supply of the CASPA tool.

7. REFERENCES

[1] ARP 4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. Aerospace Recommended Practice, Society of Automotive Engineers, Detroit, USA, 1996.

[2] S. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. In *Proc. of PDMC'03*, volume 89 of *ENTCS*, 2003.

[3] E. Böde, M. Herbstritt, H. Hermanns, S. Johr, T. Peikenkamp, R. Pulungan, R. Wimmer, and B. Becker. Compositional perfomability evaluation for statemate. In *3rd International Conference on Quantitative Evaluation of Systems (QEST)*. IEEE, 2006.

[4] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Proc. of 4th CAV*, volume 663 of *LNCS*, pages 96–108. Springer, 1992.

[5] M. Browne, E. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *Trans. on Comp.*, C-35:1035–1044, 1986.

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

[7] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proc. of VLSI'91*, pages 49–58, 1991.

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

[9] ERTMS. Project Website, May 16, 2006. `http://ertms.uic.asso.fr/etcs.html`.

[10] J.-C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic equivalence checking. In *Proc. of 5th CAV*, volume 697 of *LNCS*, pages 85–96. Springer, 1993.

[11] K. Fisler and M. Y. Vardi. Bisimulation and model checking. In *Proc. of CHARME*, LNCS, pages 338–341, 1999.

[12] J. F. Groote and F. W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. of 17th ICALP*, volume 443 of *LNCS*, pages 626–638. Springer, 1990.

[13] D. Harel and M. Politi. *Modelling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[14] M. Herbstritt, R. Wimmer, T. Peikenkamp, E. Böde, M. Adelaide, S. Johr, H. Hermanns, and B. Becker. Analysis of Large Safety-Critical Systems: A quantitative Approach. Reports of SFB/TR 14 AVACS 8, Feb 2006.

[15] H. Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *LNCS*. Springer, 2002.

[16] J.-P. Katoen et al. Faster and Symbolic CTMC Model Checking. In *Proc. of PAPM-PROBMIV*, volume 2165 of *LNCS*, pages 23–38. Springer, 2001.

[17] M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *FORTE Workshops*, volume 3236 of *LNCS*, pages 293–307. Springer, 2004.

[18] Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transition relation. In *Proc. of 30th DAC*, pages 260–265. ACM Press, 1993.

[19] R. Milner. *Communication and concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[20] R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

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

[22] R. J. van Glabbeek. The linear time – branching time spectrum II. In *Proc. of CONCUR'93*, volume 715 of *LNCS*, pages 66–81. Springer, 1993.

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

[24] R. Wimmer, M. Herbstritt, and B. Becker. Minimization of Large State Spaces using Symbolic Branching Bisimulation. In *Proc. of DDECS'06*. IEEE, 2006.

[25] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker. SIGREF – A Symbolic Bisimulation Tool Box. In *Proc. of 4th ATVA*, volume 4218 of *LNCS*, pages 477–492. Springer, 2006.