Sigref – A Symbolic Bisimulation Tool Box*

Ralf Wimmer[1], Marc Herbstritt[1], Holger Hermanns[2],
Kelley Strampp[1], and Bernd Becker[1]

[1] Albert-Ludwigs-University Freiburg, Germany
{wimmer|herbstri|strampp|becker}@informatik.uni-freiburg.de
[2] Saarland University, Saarbrücken, Germany
hermanns@cs.uni-sb.de

Abstract. We present a uniform signature-based approach to compute the most popular bisimulations. Our approach is implemented symbolically using BDDs, which enables the handling of very large transition systems. Signatures for the bisimulations are built up from a few generic building blocks, which naturally correspond to efficient BDD operations. Thus, the definition of an appropriate signature is the key for a rapid development of algorithms for other types of bisimulation.
We provide experimental evidence of the viability of this approach by presenting computational results for many bisimulations on real-world instances. The experiments show cases where our framework can handle state spaces efficiently that are far too large to handle for any tool that requires an explicit state space description.

1 Introduction

The infamous state space explosion problem is an omnipresent phenomenon in state-based verification. One promising approach to combat this problem is based on *bisimulation minimization*, where the state space is compressed by building the quotient under some appropriate notion of bisimulation. In the presence of internal activities and composition operators the benefits of this technique are particularly impressive [2–4]. The algorithmic workhorse for this minimization is a partition refinement algorithm [5, 6].

Binary decision diagrams (BDDs) are another powerful approach to handle extremely large state spaces. With BDDs such state spaces can be represented symbolically in a compact way. It is well-known that only the application of symbolic methods opened the gates for model checking of large systems [7].

This paper explores the seemingly obvious idea to combine BDDs and bisimulation minimization. This idea is not new. To our knowledge, [8, 9] were the first to apply BDD techniques to bisimulation minimization whereas Bouali [10] introduced the term "symbolic bisimulation minimization". Other recent work

* This work was partly 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.

in the context of efficient bisimulation minimization algorithms has focussed on parallel implementations, most notably the work of Blom and Orzan [11], who introduce a parallel, signature-based, branching bisimulation minimization algorithm. A *signature* is a concise characteristic function for the bisimulation.

The basic notion of bisimulation is Milner's strong bisimulation [12], which does not abstract from internal activities. In the quest for such an abstraction, very many weak bisimulation relations have been coined in the past 20 years [12–20]. Van Glabbeek's seminal overview paper [21] lists 28 different variations in the spectrum between weak and branching bisimulation, and there are many more, considering for instance similar variations for safety [20], progressing [19] and orthogonal bisimulation [17]. When it comes to applying bisimulation minimization in practice, the first question is which of the many candidate bisimulations to pick. There are some canonical ones (like ordinary weak or branching bisimulation), but certain circumstances (such as maximal progress or priority [22, 17]) may force one to opt for others. A second step is then to design an appropriate minimization algorithm for the particular choice.

We attack the second of the above problems by means of an efficient, fully symbolic and very flexible implementation of bisimulation minimization in the style of [5]. The flexibility of our algorithm stems from the fact that it is *parametric* in the *signature* used, i. e., by providing the appropriate signature, one can rapidly obtain a tailored, and efficient bisimulation minimization algorithm. For this purpose, signatures are built up from some generic building blocks, which naturally correspond to efficient BDD operations. We believe, that this approach exploits the full potential of a symbolic implementation. To validate this claim we provide experimental evidence with signatures for all the core bisimulations mentioned above. The results show that our approach can compete with the most efficient explicit algorithms but can handle much larger instances.

The paper is structured as follows. In Section 2 we introduce basic notations and definitions of the most important types of bisimulation. Additionally, Section 2 gives an overview of all considered bisimulations by presenting references to the original work as well as a discussion of algorithms that are different to the approach presented in this work. Then, in Section 3 we present our signature-based framework by stating the signatures for all bisimulations of interest. Section 4 describes the implementation of our signature-based approach that is implemented symbolically, i. e., by means of BDDs. Experimental results and a discussion of them are presented in Section 5. Finally, Section 6 concludes the paper and suggests topics for future work.

2 Preliminaries

Bisimulations typically define equivalent behavior of states in a discrete state space. In general, either state-based systems are used or transition-oriented systems like labelled transition systems. In this work, we focus on the latter.

Definition 1. *A* labelled 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 labelled transitions between states.

The usage of τ-actions depends on the application. E. g., it can serve as an abstraction mechanism to hide irrelevant actions that are internal to the system model and thus unobservable for the user. Also, in case of non-τ-actions that do not impact the property to be verified, these actions may be mapped to τ.

A bisimulation partitions the original state space into disjoint parts called *blocks* that contain those states that are equivalent regarding the applied bisimulation. It is well-known that each partition induces an equivalence relation and vice versa. Therefore, we do not distinguish between partitions and equivalence relations. We use the following notations for a partition P and an LTS $M = (S, A, T)$:

- $s \xrightarrow{a} t$ for $(s, a, t) \in T$ and $s \xrightarrow{a^*} t$ for the reflexive transitive closure of $\xrightarrow{a}$.
- $s \xrightarrow[P]{a} t$ if $s \xrightarrow{a} t$ and s and t are contained in the same block of P. Then, the transition $s \xrightarrow{a} t$ is called *inert*.
- $s \xrightarrow[P]{a^*} t$ for the reflexive transitive closure of $\xrightarrow[P]{a}$.

Bisimulations are equivalence relations on the state space of an LTS, and will be denoted in the following by $\mathfrak{B}_* \subseteq S \times S$ whereby $*$ indicates the type of the bisimulation. In the absence of τ-actions all the different notions of bisimulation considered in this work are equivalent. Otherwise, there are several ways how τ-actions can characterize the possible behavior of a state. We focus on the following bisimulations:

- Strong Bisimulation [23, 13]
- Weak Bisimulation [13, 14, 12]
- Progressing Bisimulation [19]
- Branching Bisimulation [15]
- Orthogonal Bisimulation [17]
- Delay Bisimulation [18]
- η-Bisimulation [16]
- Safety Bisimulation [20].

Strong bisimulation treats τ-actions like any other action. It is due to Park [23] and in a different formulation already to Milner [13]. Among others, it has the important property to preserve the validity of CTL* formulae and thus all interesting system properties.

Definition 2. *$\mathfrak{B}_s$ is a* strong bisimulation *if for all $s, s', t \in S$ the following holds: If $(s, t) \in \mathfrak{B}_s$ then $s \xrightarrow{a} s'$ implies that there exists $t' \in S$ with $t \xrightarrow{a} t'$ and $(s', t') \in \mathfrak{B}_s$.*

Based on the Kanelakis/Smolka algorithm [5], a symbolic algorithm for strong bisimulation has been proposed by Bouali and de Simone [10]. Dovier et al. have suggested an improvement in the form of a preprocessing step, tailored to non-strongly connected systems [24]. Since it reduces the number of iterations needed by both Bouali/de Simone's and by our algorithm in the same way we do not consider it further. There is also a symbolic $O(n \log n)$ algorithm for strong bisimulation [25] which relies on backward pointers, which are not part of popular BDD packages (e. g., CUDD [26]). Furthermore, the algorithm of Klarlund is designed for strong bisimulation only, and thus it is not obvious how to extend it to other kinds of bisimulation.

Weak Bisimulation was introduced by Milner (see [13, 14, 12]) to characterize the observable behavior of a transition system.

Definition 3. *$\mathfrak{B}_w$ is a* weak bisimulation *if for all $s, s', t \in S$ the following holds: If $(s,t) \in \mathfrak{B}_w$ then $s \xrightarrow{a} s'$ implies either $a = \tau$ and $(s',t) \in \mathfrak{B}_w$ or there exist $t', t'', t''' \in S$ with $t \xrightarrow{\tau^*} t' \xrightarrow{a} t'' \xrightarrow{\tau^*} t'''$ and $(s', t''') \in \mathfrak{B}_w$.*

A stronger version of weak bisimulation, called *progressing bisimulation*, was obtained by Montanari and Sassone [19] by requiring that sequences of τ-steps may be compressed but not omitted completely:

Definition 4. *$\mathfrak{B}_p$ is a* progressing bisimulation *if for all $s, s', t \in S$ and $a \in A$ the following holds: If $(s,t) \in \mathfrak{B}_p$ then $s \xrightarrow{a} s'$ implies that there exist $t', t'', t''' \in S$ with $t \xrightarrow{\tau^*} t' \xrightarrow{a} t'' \xrightarrow{\tau^*} t'''$ and $(s', t''') \in \mathfrak{B}_p$.*

Please note that in the definition of progressing bisimulation $a = \tau$ is allowed – even if it is an inert τ-step. This is the difference to weak bisimulation.

Branching bisimulation was introduced by van Glabbeek and Weijland [15] to overcome the problem of weak bisimulation that it does not preserve the branching structure. Branching bisimulation is comparable to stuttering equivalence on Kripke structures and preserves CTL* without next state quantifier.

Definition 5. *$\mathfrak{B}_b$ is a* branching bisimulation *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 [27].

Bergstra et al. [17] suggest a refinement of branching bisimulation called *orthogonal bisimulation*. While branching bisimulation allows sequences of τ-steps not only to be compressed but even to be omitted completely, orthogonal bisimulation does not. A state with a τ-transition cannot be orthogonally equivalent to a state without τ-transition while they may be branching equivalent.

Definition 6. *$\mathfrak{B}_o$ is an* orthogonal bisimulation *if for all $s, s', t \in S$ and $a \in A$ the following holds: If $(s,t) \in \mathfrak{B}_o$ then $s \xrightarrow{a} s'$ implies if $a \neq \tau$ then there is a $t' \in S$ with $t \xrightarrow{a} t'$ and $(s',t') \in \mathfrak{B}_o$ and if $a = \tau$ then there exist $t', t'' \in S$ with $t \xrightarrow[\mathfrak{B}_o]{\tau^*} t' \xrightarrow{\tau} t''$ and $(s', t'') \in \mathfrak{B}_o$.*

Delay bisimulation was introduced by Milner in 1981 [18].

Definition 7. *$\mathfrak{B}_d$ is a* delay bisimulation *if for all $s, s', t \in S$ the following holds: If $(s,t) \in \mathfrak{B}_d$ then $s \xrightarrow{a} s'$ implies either $a = \tau$ and $(s',t) \in \mathfrak{B}_d$ or there exist $t', t'', t''' \in S$ with $t \xrightarrow{\tau^*} t' \xrightarrow{a} t'' \xrightarrow[\mathfrak{B}_d]{\tau^*} t'''$ and $(s', t''') \in \mathfrak{B}_d$.*

The notion of *η-bisimulation* was introduced by Baeten and van Glabbeek [16].

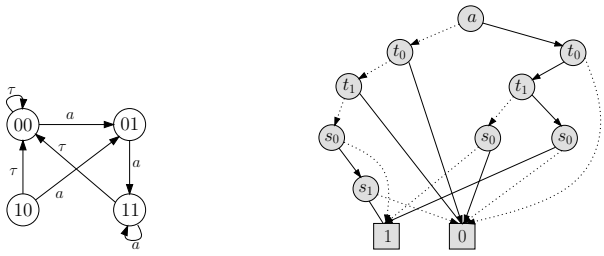

Fig. 1. An LTS and its symbolic representation

Definition 8. $\mathfrak{B}_\eta$ *is an* η-bisimulation *if for all* $s, s', t \in S$ *the following holds: If* $(s,t) \in \mathfrak{B}_\eta$ *then* $s \xrightarrow{a} s'$ *implies either* $a = \tau$ *and* $(s',t) \in \mathfrak{B}_d$ *or there exist* $t', t'', t''' \in S$ *with* $t \xrightarrow{\tau^*} t' \xrightarrow{a} t'' \xrightarrow{\tau^*} t'''$ *and* $(t,t') \in \mathfrak{B}_\eta$ *and* $(s',t''') \in \mathfrak{B}_\eta$.

Safety Bisimulation has been introduced by Bouajjani et al. in [20]. It preserves the reachability of actions, but not the branching structure of an LTS. It is useful when verifying safety properties where only reachability of states is of interest and not the way how they are reached.

Definition 9. $\mathfrak{B}_{\text{safe}}$ *is a* safety bisimulation *if for all* $s, s', s'', s''', t \in S$ *the following holds: If* $(s,t) \in \mathfrak{B}_{\text{safe}}$ *then* $s \xrightarrow{\tau^*} s' \xrightarrow{a} s'' \xrightarrow{\tau^*} s'''$ *and* $a \neq \tau$ *imply that there exist* $t', t'', t''' \in S$ *with* $t \xrightarrow{\tau^*} t' \xrightarrow{a} t'' \xrightarrow{\tau^*} t'''$ *and* $(s''', t''') \in \mathfrak{B}_{\text{safe}}$.

A key concept of our algorithm is the usage of binary decision diagrams (BDDs) [28] as a symbolic data structure for the representation of LTSs. BDDs are acyclic directed graphs that represent boolean functions over a predefined set of variables. They are obtained from binary decision trees by sharing subtrees as much as possible. By fixing the variable order on all paths from the root of the graph to a leaf, BDDs become a canonical representation of boolean functions. There exist efficient algorithms for the synthesis of BDDs. Since the mid-1980s, BDDs have become a standard data structure for automated analysis of large systems on the symbolic level. For a comprehensive treatment of BDDs and BDD algorithms, we refer to [29]. BDDs can be used for the representation of a finite set $M \subseteq \{0,1\}^n$ through its characteristic function $\chi_M : \{0,1\}^n \to \{0,1\}$ with $\chi_M(x) = 1$ iff $x \in M$. Fig. 1 shows an example of an LTS and the symbolic representation of its transition relation as a BDD. The states are encoded using two bits: The variables (s_1, s_0) are used for the present state and (t_1, t_0) for the next state of a transition. The variable a denotes the transition label with $a = 0$ denoting τ.

3 Signature-based Computation of Bisimulations

In [11], Blom and Orzan have presented a distributed explicit algorithm for the computation of *branching bisimulation*. It is based on the computation of signatures of the states. A signature $\text{sig}(s)$ can be considered as a kind of "fingerprint"

of the state $s \in S$ that characterizes reachable transitions which are relevant for the bisimulation. States with different signatures are not equivalent regarding the considered bisimulation.

Starting with the initial partition $P^0 = \{S\}$ of S, we compute for $i = 0, 1, \ldots$ a new partition by putting those states into a block that have the same signature:

$$P^{i+1} = \text{sigref}(P^i) := \{\{t \in S \mid \text{sig}(s) = \text{sig}(t)\} \,\big|\, s \in S\}$$

until a fixpoint is reached, i. e., an $n \geq 0$ with $P^n = P^{n+1}$. Using the signature for branching bisimulation as given below, Blom and Orzan were able to show that this algorithm indeed computes the coarsest branching bisimulation.

We now give signatures for all eight types of bisimulations as introduced in Section 2 (see Fig. 2 for an illustration). The proofs of correctness can be established in a similar way as in [11] for branching bisimulation. Due to page limitation, these proofs are omitted, but are contained in [30]. In the following B denotes a block of the current partition P.

- *Strong Bisimulation:*
  $\text{sig}_s(s) = \{(a, B) \mid \exists s' \in B : s \xrightarrow{a} s'\}$
- *Orthogonal Bisimulation:*
  $\text{sig}_o(s) = \{(a, B) \mid (a \neq \tau \wedge \exists t \in B : s \xrightarrow{a} t) \vee (a = \tau \wedge \exists s' \in S, s'' \in B : s \xrightarrow[P]{\tau^*} s' \xrightarrow{\tau} s'')\}$
- *Branching Bisimulation:*
  $\text{sig}_b(s) = \{(a, B) \mid \exists s' \in S, s'' \in B : s \xrightarrow[P]{\tau^*} s' \xrightarrow{a} s'' \wedge (a \neq \tau \vee (s, s'') \notin P)\}$
- *η-Bisimulation:*
  $\text{sig}_\eta(s) = \{(a, B) \mid \exists s', s'' \in S, s''' \in B : s \xrightarrow{\tau^*} s' \xrightarrow{a} s'' \xrightarrow{\tau^*} s''' \wedge (s, s') \in P \wedge (a \neq \tau \vee (s, s''') \notin P)\}$
- *Delay Bisimulation:*
  $\text{sig}_d(s) = \{(a, B) \mid \exists s' \in S, s'' \in B : s \xrightarrow{\tau^*} s' \xrightarrow{a} s'' \wedge (a \neq \tau \vee (s, s'') \notin P)\}$
- *Progressing Bisimulation:*
  $\text{sig}_p(s) = \{(a, B) \mid \exists s', s'' \in S, s''' \in B : s \xrightarrow{\tau^*} s' \xrightarrow{a} s'' \xrightarrow{\tau^*} s'''\}$
- *Weak Bisimulation:*
  $\text{sig}_w(s) = \{(a, B) \mid \exists s', s'' \in S, s''' \in B : s \xrightarrow{\tau^*} s' \xrightarrow{a} s'' \xrightarrow{\tau^*} s''' \wedge (a \neq \tau \vee (s, s''') \notin P)\}$
- *Safety Bisimulation:*
  $\text{sig}_{\text{safe}}(s) = \{(a, B) \mid \exists s', s'' \in S, s''' \in B : s \xrightarrow{\tau^*} s' \xrightarrow{a} s'' \xrightarrow{\tau^*} s''' \wedge a \neq \tau\}$

4 Symbolic Computation

We will now present how this signature-based algorithm described above can be implemented symbolically. To do so, we explain in detail the BDD representation of the LTS, the symbolic computation of the signatures, the symbolic refinement, and finally the bisimulation quotient w. r. t. a given partition of the state space.

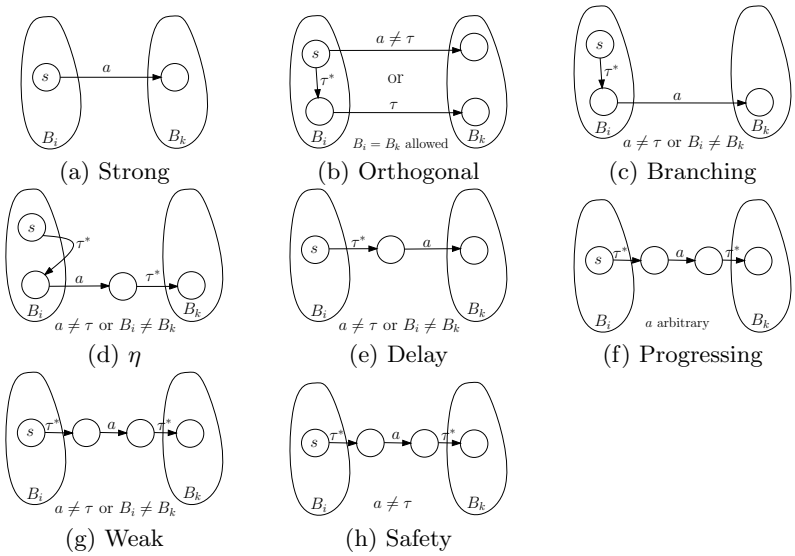

Fig. 2. Illustration of the signatures

4.1 Representation of the Data

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 assigned a unique number to each block of P (encoded using variables k) and represented P by a BDD $\mathcal{P}$ with $\mathcal{P}(s, k) = 1$ iff $s \in B_k$. All other symbolic algorithms for bisimulations 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 $\mathcal{P}$. Second, given $\mathcal{T}$ and $\mathcal{P}$, it is easy to compute the quotient w. r. t. P symbolically (see section 4.4). . We represent the signatures accordingly and create a BDD σ with $\sigma(s, a, k) = 1$ iff $(a, B_k) \in \text{sig}(s)$.

4.2 Computation of the Signatures

Now we describe the computation of the BDDs for the signatures of all eight kinds of bisimulation. For the computation we provide several "ingredients" which are listed in Table 1. The table contains a description of each operation and an expression for the BDD-based implementation.

There exist several symbolic algorithms for the computation of the reflexive transitive closure of a relation (e. g. [31]). We apply the iterative squaring method of [32] to compute $\xrightarrow[(P)]{\tau^*}$.

Operation	*BDD expression*
τ-transitions	$\mathcal{T}.Cofactor(a = \tau)$
inert τ-transitions	$\mathcal{T}.Cofactor(a = \tau) \wedge \exists k : \mathcal{P}(s,k) \wedge \mathcal{P}(t,k)$
non-τ- or non-inert transitions	$\mathcal{T}(s,a,t) \wedge \neg(inert_\tau(s,t) \wedge a \equiv \tau)$
reflexive transitive closure of $R(s,t)$	$Closure(R)$
concatenation of $R_1(s,t)$ and $R_2(s,t)$	$\exists x : R_1(s,x) \wedge R_2(x,t)$
substitute t in $R(s,t)$ by its block number	$\exists t : R(s,t) \wedge P(t,k)$

Table 1. Basic operations for the signature computation

Algorithm 1 Signature for Branching Bisimulation

1: **procedure** SigBranching
2: $inert(s,t) \leftarrow \mathcal{T}.Cofactor(a = \tau) \wedge \exists k : \mathcal{P}(s,k) \wedge \mathcal{P}(t,k)$
3: $rel(s,t) \leftarrow \mathcal{T}(s,a,t) \wedge \neg(inert(s,t) \wedge a \equiv \tau)$
4: **return** $\exists x,t : Closure(inert(s,x)) \wedge rel(x,a,t) \wedge \mathcal{P}(t,k)$

Finally we can present the algorithm for the computation of the signatures. As an example that uses all of the mentioned techniques, algorithm 1 sketches the computation of the signature for branching bisimulation.

At first, all pairs of states that are connected by an inert τ-transition are computed. In line 3 we extract all transitions that are either not inert or not labelled with τ. In the third step we put things together: the arbitrary sequence of inert τ-steps, the relevant transitions and the block numbers. The signatures for the remaining bisimulations can be computed in a similar way. Please note that everything that does not depend on the current partition, like the closure of *all* τ-steps (needed for weak, progressing, safety, η-, and delay bisimulation), can be computed as a preprocessing step.

4.3 Computation of the Refinement

We assume that we have already computed the BDD for the signatures of all states as described above. Now, we have to compute the refined partition where all states with the same signature are merged into one block.

The variable order of the BDD has to satisfy the following constraint: the s_i variables must be placed at the top of the variable order, followed by the a_j and k_l variables, i. e., $\text{level}(s_i) < \text{level}(a_j)$ and $\text{level}(s_i) < \text{level}(k_l)$ 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. To get the refined partition, we have to substitute all nodes that represent the signature of a state $s \in S$ by the BDD for the encoding of a new block number k. This is sketched in algorithm 2.

The algorithm relies on 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 are the binary encoding of a block number that has not been used in the current iteration. It is reset each time we call `refine`.

Algorithm 2 Partition Refinement

```
1: procedure REFINE(signatures σ)
2:     if σ ∈ ComputedTable then return ComputedTable[σ]
3:     x ← topVar(σ)
4:     if x = s_i then
5:         low ← Refine(σ.Cofactor(x = 0)),   high ← Refine(σ.Cofactor(x = 1))
6:         result ← returnBDDnode(x, high, low)
7:     else result ← newBlockNumber()
8:     ComputedTable[σ] ← result
9:     return result
```

Furthermore, we use 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 labelled 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.

4.4 Computation of the Quotient LTS

After we have reached the fixpoint of the signature refinement, we have to extract the bisimulation quotient. It is defined as follows:

Definition 10. *Let $M = (S, A, T)$ be a labelled transition system. Let $P = \{B_1, \ldots, B_m\}$ be a bisimulation. Then the* quotient *of M w. r. t. P (denoted M/P) is an LTS $M/P = (S_P, A_P, T_P)$ with $S_P = \{B_1, \ldots, B_m\}$, $A_P = A$, and $(B, a, B') \in T_P$ iff there are $s \in B$ and $s' \in B'$ with $(s, a, s') \in T$.*

Let $\mathcal{P}$ be a partition (represented as BDD) with $\text{sigref}(\mathcal{P}) = \mathcal{P}$. We use the notation $[k \to s]$ to denote the renaming of the k-variables to the corresponding s-variables. To extract the bisimulation quotient w. r. t. this partition, we use the block numbers as encoding for the new states: $\mathcal{S}_P = [k \to s](\exists s : \mathcal{P}(s, k))$. Then, the transition relation can be computed as follows:

$$\mathcal{R}(s, a, t) := [k \to t](\exists t : \mathcal{T}(s, a, t) \wedge \mathcal{P}(t, k))$$
$$\mathcal{T}_P(s, a, t) := [k \to s](\exists s : \mathcal{R}(s, a, t) \wedge \mathcal{P}(s, k))$$

4.5 Improvements

During our experiments we observed that the BDD for the expression $\exists k : \mathcal{P}(s, k) \wedge \mathcal{P}(t, k)$, which is used for the computation of the inert τ-transitions, is considerably larger than the BDD for $\mathcal{P}(s, k)$. This expensive step can be avoided by computing the signatures and the refinement only *for one block at a time*. To do so, the function SIG gets an additional parameter for the states for which we have to compute the signatures. Then, a transition is inert iff the

source state as well as the target state are contained in this block. We apply this technique to all bisimulations where the signature depends on the current partition (this is not the case for strong, safety, and progressing bisimulation).

The sequential refinement enables us to apply a dedicated optimization 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 remaining blocks this information can be used already in the same iteration. Block forwarding substantially reduces the number of iterations to the fixpoint. Both techniques result in a large speedup for almost all of our examples.

5 Experimental Results

We have implemented our approach in a tool, called SIGREF, that relies on the popular BDD-package CUDD [26]. For comparison, we also implemented the strong bisimulation algorithm presented by Bouali/de Simone in [10]. Additionally, we extended Bouali/de Simone's algorithm to weak and branching bisimulation, as it was briefly suggested in their paper. We were also able to extend Bouali/de Simone's algorithm to safety bisimulation. For comparison with bisimulation tools requiring an explicit state space representation, we use BCGMIN [33] which is part of the protocol verification toolbox CADP [34].

For the evaluation, we use examples stemming from two quite different domains: compositional process algebraic system descriptions and STATEMATE designs that are extended by failure-behavior. Regarding the meaning of the τ-action, for process algebraic descriptions τ is typically used to hide synchronization of the involved components. Our STATEMATE descriptions are designed to allow a quantitive analysis of the malfunctioning of the system, and therefore nominal non-failure-actions are exchanged by the τ-action, since only failure-actions are of interest. In [35] you will find more about our approach for quantitative analysis of STATEMATE designs.

Kanban production system. Here, we use a process-algebraic description of a Kanban system [36] 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 [37]. 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.

Table 2 shows details for the generated LTSs as well as the size of the bisimulation quotient for all considered bisimulations. $|S|$ ($|T|$) denotes the number of states (transitions), respectively. For entries denoted with 'n. a.', none of the algorithms, i. e., Bouali/de Simone, BCGMIN, or SIGREF, were able to compute the bisimulation quotient. All bisimulations result in impressive reductions of the state space. E. g., for 8 workpieces, branching bisimulation reduces $|S|$ by a factor of nearly 62.000, and $|T|$ by a factor of about 82.000.

p		*Original*	*Strong*	*Orthogonal*	*Branching*	η	*Delay*	*Progressing*	*Weak*	*Safety*
1	$\|S\|$	256	148	52	24	24	24	52	24	24
	$\|T\|$	904	472	111	42	42	42	111	42	42
2	$\|S\|$	63772	5725	1005	206	206	206	561	206	206
	$\|T\|$	231424	30860	3556	552	552	552	1869	552	552
3	$\|S\|$	1024240	85356	8838	872	872	872	2643	872	872
	$\|T\|$	4651520	601650	40708	2968	2968	2968	11015	2968	2968
4	$\|S\|$	16020316	778485	51805	2785	2785	2785	8964	2785	2785
	$\|T\|$	74424320	6419550	278059	10932	10932	10932	42576	10932	10932
5	$\|S\|$	16772032	5033631	n. a.	7366	7366	7366	24643	7366	7366
	$\|T\|$	133938560	46071311	n. a.	31795	31795	31795	127604	31795	31795
6	$\|S\|$	264515056	n. a.	n. a.	17010	17010	17010	58463	17010	17010
	$\|T\|$	1689124864	n. a.	n. a.	78584	78584	78584	321931	78584	78584
7	$\|S\|$	268430272	n. a.	n. a.	35456	35456	35456	124311	35456	35456
	$\|T\|$	2617982976	n. a.	n. a.	172382	172382	172382	716829	172382	172382
8	$\|S\|$	4224876912	n. a.	n. a.	68217	68217	68217	242858	68217	68217
	$\|T\|$	29070458880	n. a.	n. a.	345128	345128	345128	1451590	345128	345128

Table 2. Size of the LTS for the Kanban system with different number of workpieces

		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Strong	SIGREF	0.01	2.23	93.77	1814.81	22862.70	ML	ML	ML
	bouali	0.07	152.69	13110.80	TL	TL	TL	TL	TL
	BCGMIN	0.14	1.36	99.08	2335.86	18164.83	ML	ML	ML
Orthogonal	SIGREF	0.01	6.02	388.79	16836.10	TL	TL	TL	TL
Branching	SIGREF	0.01	0.51	12.13	107.90	617.71	2685.59	15020.50	53725.40
	bouali	0.01	0.12	0.93	5.71	25.33	77.15	770.83	141591.00
	BCGMIN	0.21	0.51	10.01	193.25	559.89	ML	ML	ML
η	SIGREF	< 0.01	0.24	4.73	42.19	219.32	946.48	6636.97	22743.90
delay	SIGREF	< 0.01	0.18	3.44	33.28	183.13	806.34	4736.58	13206.90
Progressing	SIGREF	< 0.01	0.14	1.44	9.55	69.17	347.21	2400.08	5824.31
Weak	SIGREF	< 0.01	0.19	3.63	33.91	173.86	773.24	5711.58	14970.10
	bouali	< 0.01	0.11	0.89	5.49	25.05	71.83	622.29	146971.00
Safety	SIGREF	< 0.01	0.03	0.25	1.49	5.78	21.62	120.22	543.96
	bouali	0.01	0.12	0.90	5.48	27.95	71.92	709.25	140730.00

Table 3. CPU runtimes of the three tools applied to the Kanban benchmark

Table 3 gives the runtimes[3] of all three algorithms, i. e., the explicit tool BCGMIN, Bouali/de Simone's BDD algorithm, and our signature-based approach SIGREF. Please note: BCGMIN only provides strong and branching bisimulation. Typically, algorithms that use an explicit state space representation are faster than symbolic ones. Therefore, it is very interesting that SIGREF is competitive to BCGMIN for both strong and branching bisimulation. However, in contrast to BCGMIN, SIGREF is able to handle branching bisimulation for very large instances, i. e., for 5 workpieces or more. Compared to the algorithm of Bouali/de Simone, SIGREF performs much more efficient, in particular for large instances. Except strong and orthogonal bisimulation, SIGREF is able to compute all remaining kinds of bisimulation completely. Clearly, we have to admit that for large instances SIGREF requires a huge amount of time. However, these bisimulations cannot be computed by either BCGMIN or Bouali/de Simone's algorithm. As a summary, the results of Table 3 show that SIGREF can efficiently handle a large variety of bisimulations. Even compared to explicit state algorithms, SIGREF performs very competitive, and from an application point of view much more robust.

[3] All experiments in this work were performed on an AMD Opteron 2.4 GHz CPU. We have set a time limit of 160.000 seconds and a main memory limit of 2 GB. Entries "TL" and "ML" mean that the time or memory limit was exceeded, respectively.

Input		*Original*	*Strong*	*Orthog.*	*Branch.*	η	*Delay*	*Progress.*	*Weak*	*Safety*
etcs-1	$\|S\|$	1057	51	51	51	50	50	50	50	50
	$\|T\|$	15058	749	749	749	731	731	731	731	1172
etcs-2	$\|S\|$	428113	1312	1312	1312	1154	1214	1102	1102	1102
	$\|T\|$	16589262	48848	48848	48848	42291	45352	40540	40540	71298
etcs-3	$\|S\|$	158723041	35842	35842	35842	30173	31999	28451	28451	28451
	$\|T\|$	16658393318	3128876	3128876	3128876	2628447	2808983	2492665	2492665	4459877
bs-p	$\|S\|$	184865921	1469	1469	1177	856	951	847	847	847
	$\|T\|$	10025344274	60483	60483	42830	31970	36351	31700	31700	165312
ctrl	$\|S\|$	139623	14614	14615	9627	8077	8093	7427	7427	7427
	$\|T\|$	11867888	1033582	1033582	653303	523989	525402	482866	482866	1005666

Table 4. Size of the STATEMATE benchmarks

Failure-enhanced STATEMATE *descriptions.* As a second benchmark suite we analyzed LTSs that were generated from STATEMATE descriptions [38] that were 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 [39] 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 [40] that models a braking system from an airplane. It is about the correctness of the pilot's braking pedal and the hydraulic pressure given to the wheels of the airplane. The benchmark *ctrl* describes a redundancy controller of an industrial avionics project. A detailed description of all benchmarks can be found in [35].

Table 4 shows for each STATEMATE example the size of the LTS and of the corresponding quotient, depending on the applied bisimulation. Table 5 gives the CPU runtimes for the different algorithms. The dominant performance of SIGREF is obvious. Only the computation of branching bisimulation for the *ctrl* example shows an advantage of BCGMIN. However, SIGREF is still able to handle this example. A rough estimate of the CPU runtimes of Bouali/de Simone's algorithm shows that it performs two orders of magnitude worse than SIGREF. And for both examples *bs-p* and *ctrl*, the algorithm of Bouali/de Simone is not able to compute any of the provided bisimulations. Therefore, Table 4 again shows, but now in a much more impressive manner, that SIGREF is not only able to outperform existing approaches, but that it is applicable to a wider range of applications.

To get an insight why Bouali/de Simone's algorithm performs so badly, we had a detailed look at the CPU runtimes and the size of the BDDs for the representation of the partition during the iterative refinement. Figure 3 shows the corresponding data for SIGREF and the algorithm of Bouali/de Simone, respectively. The left y-axis denotes the CPU runtime and the right y-axis depicts the size of the BDD for the partition (in logarithmic scale). The x-axis corresponds to the iterations during the refinement. It it obvious that the BDD size is much more moderate for SIGREF. This directly impacts the CPU runtime. The difference between SIGREF and Bouali/de Simone's algorithm is that SIGREF relies on a predicate $\mathcal{P}(s,k)$ for storing the information that state s is contained in block k. The algorithm of Bouali/de Simone, however, uses a predicate $\mathcal{P}(s,t)$ denoting

		etcs-1	*etcs-2*	*etcs-3*	*bs-p*	*ctrl*
Strong	SIGREF	0.04	8.96	958.93	21.44	106.73
	bouali	0.20	1880.16	82749.40	TL	TL
	BCGMIN	0.27	16.27	ML	ML	848.94
Orthogonal	SIGREF	0.08	49.56	16706.20	348.85	3849.29
Branching	SIGREF	0.06	49.76	20912.00	276.78	1701.22
	bouali	0.31	2594.10	98897.90	TL	TL
	BCGMIN	0.28	22.63	ML	ML	378.55
η	SIGREF	0.18	133.59	16162.10	25992.50	1124.90
Delay	SIGREF	0.08	75.63	16336.60	1328.60	1026.80
Progressing	SIGREF	0.09	43.59	2177.10	13739.50	81.03
Weak	SIGREF	0.12	99.55	13434.40	13938.40	956.00
	bouali	0.43	4340.91	113336.00	TL	TL
Safety	SIGREF	0.11	42.62	2214.39	16653.60	76.29
	bouali	0.38	4383.46	112802.00	TL	TL

Table 5. Runtimes for the STATEMATE benchmarks

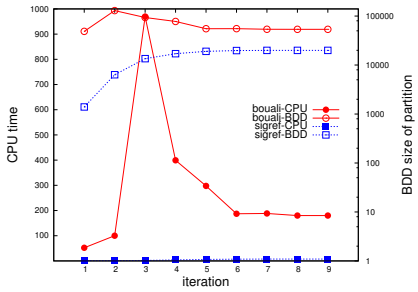

Fig. 3. Bouali/de Simone vs. SIGREF for branching bisimulation on the *etcs2* example.

that state s and state t are contained in the same block. The advantage of SIGREF's predicate $\mathcal{P}(s,k)$ seems to be the *sharing* of the block number k, i. e., the signature refinement algorithm only needs to efficiently decide whether there are multiple states in a block k, but it is enough to *implicitly* store the information which states are in the same block. Put another way, the inherent symmetry of the predicate $\mathcal{P}'(s,t)$ of Bouali/de Simone's algorithm, i. e., $\mathcal{P}'(s,t) \Leftrightarrow \mathcal{P}'(t,s)$, is more than needed for our signature-based approach. This information overhead results in huge BDDs, which consequently leads to bad runtimes.

6 Conclusion and Future Work

In this work, we have presented a uniform and easily extendible framework for the computation of several kinds of bisimulation. We have evaluated our approach on examples from process algebra as well as from STATEMATE descriptions. Furthermore, we compared our algorithm to other state-of-the-art algorithms.

Our experiments show that in almost all cases our implementation SIGREF can handle much larger systems than other algorithms, thereby requiring less time. We found that the algorithm of Bouali/de Simone suffers from the redundant representation of partitions. On the other hand, SIGREF gains from dedicated optimizations, e. g. block forwarding. The experiments clearly show that the signature-based approach coupled with BDDs outperforms other state-of-the-art algorithms with respect to (1) the size of the system under analysis, (2) the variety of applicable models, and (3) the CPU runtimes.

As future work, we will check whether SIGREF can be extended by some input language for signatures such that new types of bisimulation can be defined without significant programming effort. Furthermore, we are investigating how the signature-based approach can be extended to compute stochastic bisimulations defined on Interactive Markov Chains (IMCs) [41].

Acknowledgments We would like to thank the whole AVACS::S3 team for its fruitful cooperation. Especially, we'd like to 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.

References

1. Wimmer, R., Herbstritt, M., Becker, B.: Minimization of Large State Spaces using Symbolic Branching Bisimulation. In: Proc. of IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS). (2006) 9–14
2. Chehaibar, G., et al.: Specification and Verification of the PowerScale^{TM} Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In: Proc. of FORTE. Volume 69. (1996) 435–450
3. Giannakopoulou, D.: Model Checking for Concurrent Software Architectures. PhD thesis, Imperial College, University of London (1999)
4. Graf, S., Steffen, B., Luttgen, G.: Compositional minimisation of finite state systems using interface specifications. Formal Asp. of Comp. **8**(5) (1996) 607–616
5. Kanellakis, P., Smolka, S.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation **86**(1) (1990) 43–68
6. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Jour. on Computing **16**(6) (1987) 973–989
7. Burch, J., et al.: Symbolic Model Checking: 10^{20} States and Beyond. Information and Computation **98**(2) (1992) 142–170
8. Bouajjani, A., Fernandez, J.C., Halbwachs, N.: Minimal model generation. In: Proc. of CAV. Volume 531 of LNCS., Springer (1991) 197–203
9. Bouajjani, A., Fernandez, J.C., Halbwachs, N., Ratel, C., Raymond, P.: Minimal state graph generation. Science of Computer Programming **18** (1992) 247–269
10. Bouali, A., de Simone, R.: Symbolic Bisimulation Minimisation. In: Proc. of CAV. Volume 663 of LNCS., Springer (1992) 96–108
11. Blom, S., Orzan, S.: Distributed Branching Bisimulation Reduction of State Spaces. ENTCS **89**(1) (2003) 990–113
12. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
13. Milner, R.: A Calculus of Communicating Systems. Volume 92 of LNCS. (1980)
14. Milner, R.: Lectures on a Calculus for Communicating Systems. In: Proc. Seminar on Concurrency. Volume 197 of LNCS., Springer (1984) 197–220
15. van Glabbeek, R., Weijland, W.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM **43**(3) (1996) 555–600
16. Baeten, J., van Glabbeek, R.: Another Look at Abstraction in Process Algebra. In: Proc. of ICALP. Volume 267 of LNCS., Springer (1987) 84–94
17. Bergstra, J.A., Ponse, A., van der Zwaag, M.B.: Branching time and orthogonal bisimulation equivalence. Theor. Comp. Sci. **309** (2003) 313–355
18. Milner, R.: A Modal Characterization of Observable Machine-Behaviour. In: Proc. of CAAP. Volume 112 of LNCS., Springer (1981) 25–34

19. Montanari, U., Sassone, V.: Dynamic congruence vs. progressing bisimulation for CCS. Fundam. Inform. **16**(1) (1992) 171–199
20. Bouajjani, A., et al.: Safety for Branching Time Semantics. In: Proc. of ICALP. Volume 510 of LNCS., Springer (1991) 76–92
21. van Glabbeek, R.J.: The linear time – branching time spectrum II. In: Proc. of CONCUR. Volume 715 of LNCS., Springer (1993) 66–81
22. Hermanns, H., Lohrey, M.: Priority and maximal progress are completely axiomatisable. In: Proc. of CONCUR. Volume 1466 of LNCS., Springer (1998) 237–252 (Extended Abstract).
23. Park, D.: Concurrency and automata on infinite sequences. In: GI Conf. on Theor. Comp. Sci. Volume 104 of LNCS., Springer (1981) 167–183
24. Dovier, A., Gentilini, R., Piazza, C., Policriti, A.: Rank-based symbolic bisimulation (and model checking). ENTCS **67** (2002)
25. Klarlund, N.: An $n \log n$ algorithm for online BDD refinement. In: Proc. of CAV. Volume 1254 of LNCS., Springer (1997) 107–118
26. Somenzi, F.: CUDD: CU Decision Diagram Package Release 2.4.1. University of Colorado at Boulder (2005)
27. Groote, J.F., Vaandrager, F.W.: An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In Paterson, M., ed.: Proc. of ICALP. Volume 443 of LNCS., Springer (1990) 626–638
28. Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. on Comp. **35**(8) (1986) 677–691
29. Wegener, I.: Branching programs and binary decision diagrams. SIAM Monographs on Discrete Mathematics and Applications. SIAM (2000)
30. Strampp, K.: Symbolische Berechnung von Bisimulationen. Diploma thesis, Albert-Ludwigs-University Freiburg, Germany (2006)
31. Matsunaga, Y., McGeer, P.C., Brayton, R.K.: On computing the transitive closure of a state transition relation. In: Proc. of DAC, ACM Press (1993) 260–265
32. Burch, J.R., et al.: Sequential circuit verification using symbolic model checking. In: Proc. of DAC, ACM Press (1990) 46–51
33. Garavel, H., Hermanns, H.: On Combining Functional Verification and Performance Evaluation using CADP. In: Proc. of FME. Volume 2391 of LNCS. (2002)
34. Fernandez, J.C., et al.: CADP: A Protocol Validation and Verification Toolbox. In: Proc. of CAV. Volume 1102 of LNCS. (1996) 437–440
35. Herbstritt, M., Wimmer, R., Peikenkamp, T., Böde, E., Adelaide, M., Johr, S., Hermanns, H., Becker, B.: Analysis of Large Safety-Critical Systems: A quantitative Approach. Reports of SFB/TR 14 AVACS 8 (2006) ISSN: 1860-9821.
36. Ciardo, G., Tilgner, M.: On the use of Kronecker operators for the solution of generalized stochastic Petri nets. Technical Report 96-35, ICASE (1996)
37. Kuntz, M., Siegle, M., Werner, E.: Symbolic Performance and Dependability Evaluation with the Tool CASPA. In: FORTE Workshops. Volume 3236 of LNCS., Springer (2004) 293–307
38. Harel, D., Politi, M.: Modelling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw-Hill (1998)
39. ERTMS: Project Website (May 16, 2006) `http://ertms.uic.asso.fr/etcs.html`.
40. 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)
41. Hermanns, H.: Interactive Markov Chains: The Quest for Quantified Quality. Volume 2428 of LNCS. Springer (2002)