Simulation Revisited

Li Tan and Rance Cleaveland

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11790 USA
{tanli, rance}@cs.sunysb.edu

Abstract. This paper develops an efficient algorithm for determining when one system is capable of simulating the behavior of another. The method combines an iterative algorithm for computing behavioral preorders with an algorithm that simultaneously computes the bisimulation equivalence classes of the systems in question. Experimental data indicate that the new routine dramatically outperforms the best-known algoritm for computing simulation, even when the systems are minimized with respect to bisimulation before the simulation algorithm is invoked.

1 Introduction

A traditional problem in the verification of concurrent systems is the following: given two processes A and B, does B *simulate* A [Mil71]? The resulting *simulation ordering* has numerous practical motivations, both in its own right as a *refinement / approximation ordering* [BBLS92,DGG97,Jon91,LV95] and as a vehicle on which to base the definitions of other refinement orderings [BHR84,DNH83]. Indeed, efficient algorithms for computing the simulation ordering underpin algorithms for computing relations such as trace inclusion and the failures/must preorder [CH93].

Despite its utility, not much attention has been paid to algorithms for computing the simulation ordering for finite-state systems. Bloom and Paige [BP95] present a global routine that runs in time $O(m_1 n_2 + m_2 n_1)$, where m_i and n_i represent the number of states and transitions in the two systems being checked. Essentially the same algorithm was discovered independently in [HHK95], and similar ideas may be found in [CC95,CS90]. Celikkan [Cel95] defines an on-the-fly algorithm of comparable complexity.

This paper develops a new technique for computing the simulation ordering that combines the fixpoint calculation techniques of [BP95] with the fast *bisimulation-minimization* algorithm due to Paige and Tarjan [PT87,Fer90]. One well-known way to improve the performance of a simulation checker is first to minimize the systems in question with respect to bisimulation in order to reduce the number of states that must be considered. By intertwining the computation of the bisimulation equivalence classes with the simulation relation, our approach exploits the benefits of minimization while avoiding the complete computation of equivalence classes if this is unnecessary.

2 Background

In this paper systems will be modeled as *labeled transition systems* (LTSs).

Definition 1. *A* labeled transition system *is a triple $\langle S, A, \longrightarrow \rangle$, where S is a set of states, A a set of actions, and $\longrightarrow \subseteq S \times A \times S$ the transition relation.*

States may be seen as "configurations" the system may enter, while actions represent system activities that can cause state changes. We write $s \xrightarrow{a} s'$ in lieu of $\langle s, a, s' \rangle \in \longrightarrow$, and we sometimes abuse terminology by referring to a tuple $\langle S, A, \longrightarrow, s_I \rangle$, where $s_I \in S$ is the *start state*, as a labeled transition system.

As we make extensive use of binary relations, we introduce some terminology here. If $R \subseteq S \times S$ is a binary relation over set S then we write $R^{-1} = \{\langle s', s \rangle \mid \langle s, s' \rangle \in R\}$ for the inverse of R. Also, if $s \in S$ then we use $R(s)$ to represent the set $\{s' \mid \langle s, s' \rangle \in R\}$, and if $T \subseteq S$ we define $R(T) = \bigcup_{s \in T} R(s)$.

Definition 2. *Let $\langle S, A, \longrightarrow \rangle$ be a LTS, and let $R \subseteq S \times S$ be a relation. Then:*

1. *R is a* simulation *if for every $\langle s_1, s_2 \rangle \in R$ and $a \in A$, whenever $s_1 \xrightarrow{a} s_1'$ then there is a s_2' such that $s_2 \xrightarrow{a} s_2'$ and $\langle s_1', s_2' \rangle \in R$.*
2. *R is a* bisimulation *if both R and R^{-1} are simulations.*

It is easy to establish that for any LTS there is a maximal simulation, $\preceq$, and bisimulation, $\sim$, and that the former is a preorder while the latter is an equivalence relation. The following states an obvious connection between $\preceq$ and $\sim$.

Theorem 1. *Let $\langle S, A, \longrightarrow \rangle$ be a LTS, with $s_1, s_2, s_3 \in S$. Then:*

1. *If $s_1 \sim s_2$ and $s_2 \preceq s_3$ then $s_1 \preceq s_3$.*
2. *If $s_1 \preceq s_2$ and $s_2 \sim s_3$ then $s_1 \preceq s_3$.*

This result has practical implications for computing $\preceq$, since it indicates that LTSs may be minimized with respect to $\sim$ before calculating $\preceq$.

The notion of simulation may be extended to two LTSs as well. Let $T_1 = \langle S, A_1, \longrightarrow_1 \rangle$ and $T_2 = \langle S_2, A_2, \longrightarrow_2 \rangle$ be LTSs. Then a *simulation from T_1 to T_2* is relation $R \subseteq S_1 \times S_2$ satisfying the following for every $\langle s_1, s_2 \rangle \in R$ and $a \in A$:

if $s_1 \xrightarrow{a}_1 s_1'$ then there is a s_2' such that $s_2 \xrightarrow{a}_2 s_2'$ and $\langle s_1', s_2' \rangle \in R$.

A maximal simulation $\preceq$ from T_1 to T_2 exists, and if $s_1 \in S_1$ and $s_2 \in S_2$ then we write $s_1 \preceq s_2$ when these states are in this relation. If $S_1 \cap S_2 = \emptyset$ then it is easy to show that $s_1 \preceq s_2$ in the sense just described if and only if s_1 and s_2 are related by the maximal simulation in the single transition systems $T = \langle S_1 \cup S_2, A_1 \cup A_2, \longrightarrow_1 \cup \longrightarrow_2 \rangle$. If $T_1 = \langle S_1, A_1, \longrightarrow_1, s_1 \rangle$ and $T_2 = \langle S_2, A_2, \longrightarrow_2, s_2 \rangle$ have start states s_1 and s_2 indicated, then we write $T_1 \preceq T_2$ if $s_1 \preceq s_2$.

3 The Relational Coarsest KA-Partition Problem

We are interested in determining algorithmically whether $T_1 \preceq T_2$, where $T_1 = \langle S_1, A, \longrightarrow_1, s_1\rangle$ and $T_2 = \langle S_2, A, \longrightarrow_2, s_2\rangle$ are LTSs. To simplify the presentation, we consider a restricted version of the problem in which transition systems are *unlabeled*. In what follows a(n unlabeled) transition system is a pair $\langle S, E\rangle$ where S is a set of states and $E \subseteq S \times S$ is the transition relation. On occasion, we designate a start state $s_I \in S$ and call $\langle S, E, s_I\rangle$ a transition system. Transition systems may be seen as labeled transition systems whose action set A contains a single action. Note that in $\langle S, E\rangle$ E is a binary relation over S.

In this section we show how calculating $\preceq$ can be reduced to solving the *Relational Coarsest KA-Partition Problem.* To define this problem, we first review the Relational Coarsest Partition Problem [PT87], whose solution corresponds to computing the equivalence classes of $\sim$ over a single transition system.

The Relational Coarsest Partition Problem (RCPP). The statement of the RCPP uses the following terminology.

Definition 3. *Let $\langle S, E\rangle$ be a transition system.*

1. *A* partition *of S is a collection $\{B_1, \ldots, B_n\}$ of disjoint nonempty subsets of S such that $S = \bigcup_{i=1}^{n} B_i$. Each B_i in a partition P is called a* block *of P.*
2. *A partition P* refines *a partition P' ($P \trianglelefteq P'$) if for every $B_i \in P$ there is a $B'_j \in P'$ such that $B_i \subseteq B'_j$. In this case we say that P' is* coarser *than P.*
3. *Let $S_1, S'_1 \subseteq S$. Then S_1 is* stable *with respect to S'_1 if either $S_1 \cap E^{-1}(S'_1) = \emptyset$ or $S_1 \subseteq E^{-1}(S'_1)$. A partition P of S is stable with respect to a set $S' \subseteq S$ if each $B_i \in P$ is stable with respect to S'. A partitition P is stable with respect to another partition P' if every $B_i \in P$ is stable with respect to every $B'_j \in P'$. A partition P is* self-stable *if it is stable with respect to itself.*

Intuitively, a set S_1 is stable with respect to S'_1 if either no state in S_1 has a transition into S'_1 ($S_1 \cap E^{-1}(S'_1) = \emptyset$) or every state in S_1 has at least one transition into S'_1 ($S_1 \subseteq E^{-1}(S'_1)$). It is easy to see that $\trianglelefteq$ defines a partial order over the set of partitions of S and that a coarsest self-stable partition of S is guaranteed to exist. The RCPP may now be defined as follows.

Given: Transition system $\langle S, E\rangle$ with $|S| < \infty$.
Compute: The coarsest self-stable partition P of S.

One may show that any self-stable partition of S is a bisimulation and that the blocks in the largest self-stable partition of S are the equivalence classes of $\sim$.

The Relational Coarsest KA-Partition Problem. Theorem 1 suggests one way to improve the efficiency of computing whether or not $T_1 \preceq T_2$: minimize both T_1 and T_2 with respect to $\sim$ before calculating $\preceq$ using e.g. the algorithm in [BP95]. Doing so entails using a preprocessing step to compute the equivalence classes of $\sim$ for each of T_1 and T_2. Our goal is to find an way to compute bisimulation classes and a simulation relation simultaneously, thereby eliminating the need for fully computing equivalence classes when this is unnecessary. Our method involves associating *auxiliary information* in the form of a set of "potentially simulating states" with each equivalence class of states in the "lower" transition system. This auxiliary information will also be recorded in terms of equivalence classes of states in the "upper" transition system. When a lower equivalence class is split, auxiliary information must be altered appropriately. To make these notions precise, we define the *Relational Coarsest KA-Partition Problem*, which is an alteration of the RCPP introduced above.

Definition 4. *Let $T_1 = \langle S_1, E_1\rangle$ and $T_2 = \langle S_2, E_2\rangle$ be transition systems.*

1. *A* kernel-auxiliary pair *(KA-pair) has form $\langle B, X\rangle$, where $B \subseteq S_1$ and $X \subseteq S_2$. We write $\langle B, X\rangle \subseteq \langle B', X'\rangle$ if $B \subseteq B'$ and $X \subseteq X'$. We often refer to B as the* kernel set *of $\langle B, X\rangle$ and X as the* auxiliary set.
2. *A set P of KA-pairs is a* kernel-auxiliary partition *(KA-partition) from T_1 to T_2 if $P_1 = \{B \mid \langle B, X\rangle \in P\}$ is a partition of S_1.*
3. *A KA-partition P* refines *KA-partition P' ($P \trianglelefteq P'$) if for every $\langle B, X\rangle \in P$ there is a $\langle B', X'\rangle \in P'$ such that $\langle B, X\rangle \subseteq \langle B', X'\rangle$.*
4. *KA-pair $\langle B, X\rangle$ is* stable *with respect to KA-pair $\langle B', X'\rangle$ if either $B \cap E_1^{-1}(B') = \emptyset$, or $B \subseteq E_1^{-1}(B')$ and $X \subseteq {E_2}^{-1}(X')$. A KA-partition P is stable with respect to KA-pair $\langle B', X'\rangle$ if every $\langle B, X\rangle \in P$ is stable with respect to $\langle B', X'\rangle$. KA-partition P is stable with respect to KA-partition P' if P is stable with respect to every KA-pair in P'. KA-partition P is self-stable if it is stable with respect to itself.*

Note that if P is a self-stable KA-partition from T_1 to T_2 and $\langle B, X\rangle, \langle B', X'\rangle \in P$, then either no state in B has a transition into B', or every state in B has a transition into B' and every state in X has a transition into X'. When P is a self-stable KA-partition, the set $\{B \mid \langle B, X\rangle \in P\}$ is a self-stable partition.

Every KA-partition P defines a relation $R(P) \subseteq S_1 \times S_2$ given by: $\langle s_1, s_2\rangle \in R(P)$ if and only if there exists $\langle B, X\rangle \in P$ such that $s_1 \in B$ and $s_2 \in X$. The following is an easy consequence of Definition 4 and Theorem 1.

Theorem 2. *Let $T_1 = \langle S_1, E_1\rangle$ and $T_2 = \langle S_2, E_2\rangle$ be transition systems with $s_1 \in S_1$ and $s_2 \in S_2$. Then there is a unique coarsest self-stable KA-partition $P_{\max}$ from T_1 to T_2, and $s_1 \preceq s_2$ if and only if $\langle s_1, s_2\rangle \in R(P_{\max})$.*

The Relational Coarsest KA-Partition Problem may now be stated as follows.

Given: Transition systems $T_1 = \langle S_1, E_1\rangle$, $T_2 = \langle S_2, E_2\rangle$, with $|S_1|, |S_2| < \infty$.
Compute: The coarsest self-stable partition P from T_1 to T_2.

The statement of this problem does not mention partitions of the state set of the "upper" transition system. The following corollary indicates that auxiliary sets can be efficiently represented as unions of bisimulation-equivalence classes.

Corollary 1. *Let $T_1 = \langle S_1, E_1\rangle$ and $T_2 = \langle S_2, E_2\rangle$ be transition systems, let $P_{\max}$ be the coarsest self-stable KA-partition on T_1 and T_2. and let Q be the coarsest self-stable partition over S_2. Then for any $\langle B, X\rangle \in P_{\max}$ and $C \in Q$, either $X \subseteq C$ or $X \cap C = \emptyset$.*

4 Computing the Relational Coarsest KA-Partition

This section presents two approaches to constructing the relational coarsest KA-partition on two systems. The first is based on the “naive” relational coarsest partition algorithm of [PT87], and we include it here to illustrate the basic operations that both algorithms must perform. The second builds on the sophisticated “three-way splitting” approach also found in [PT87].

4.1 A Naive Approach

The naive algorithm uses a *partition-refinement* strategy. Starting with the coarsest possible KA-partition, KA-pairs are repeatedly “split” until the KA-partition becomes self-stable. The basic operation in the algorithm involves splitting the KA-pairs in a KA-partition P with respect to a KA-pair $C' = \langle B', X' \rangle$.

---

split$(C' = \langle B', X' \rangle, P')$
1 $\quad P' := \emptyset$
2 $\quad$ **for** every $\langle B, X \rangle \in P$ **do**
3 $\quad\quad$ **if** $B \cap E_1^{-1}(B') \neq \emptyset$ **then**
4 $\quad\quad\quad$ **if** $B \not\subseteq E_1^{-1}(B')$ **then**
5 $\quad\quad\quad\quad \langle B_1, X_1 \rangle := \langle B \cap E_1^{-1}(B'), X \cap E_2^{-1}(X') \rangle$
6 $\quad\quad\quad\quad \langle B_2, X_2 \rangle := \langle B - E_1^{-1}(B'), X \rangle$
7 $\quad\quad\quad\quad P' := P' \cup \{\langle B_1, X_1 \rangle, \langle B_2, X_2 \rangle\}$
8 $\quad\quad\quad$ **else**
9 $\quad\quad\quad\quad \langle B_1, X_1 \rangle := \langle B, X \cap E_2^{-1}(X') \rangle$
10 $\quad\quad\quad\quad P' := P' \cup \{\langle B_1, X_1 \rangle\}$
11 $\quad\quad$ **else** $P' := P' \cup \{\langle B, X \rangle\}$
return(P')

The crucial insight underlying this operation occurs in lines 5–6. In this situation the kernel set, B, of KA-pair $\langle B, X \rangle$ must be split because it is not stable with respect to B', the kernel set of C'. Because states in B_1 have transitions into B', the “auxiliary set”, X_1, of states that potentially simulate those in B_1 must also have transitions into X'. The auxiliary set X_2 of B_2 does not have to satisfy this requirement, since states in B_2 do not have transitions into B'. Note that both $\langle B_1, X_1 \rangle$ and $\langle B_2, X_2 \rangle$ are stable with respect to C'.

We call C' a *splitter* for P if $P \neq$ *split*(C', P) (note this means that P is not stable with respect to C'). The naive algorithm works as follows.

$P := \{\langle S_1, S_2 \rangle\}$
while P is not stable with respect to some $C' = \langle B', X' \rangle \in P$ **do**
$\quad P :=$ *split*(C', P)

It will be convenient in what follows to view the execution of our KA-partition algorithms in terms of a tree whose nodes are labeled with KA-pairs. A node has children if the KA-pairs labeling the children are the result of applying a *split* operation to the label of the node. Thus a node may have two children (if its kernel set is split in lines 5–6) or one child (if its kernel set remains unchanged but its auxiliary set is *pruned* in line 9). If a node labeled by $\langle B, X\rangle$ has two children, we assume the left child is labeled by $\langle B_1, X_1\rangle$ (line 6) and the right by $\langle B_2, X_2\rangle$. An invariant in this tree is that the right child of every two-child node has the same auxiliary set as its parent. We refer to such a tree as a *partition tree.* Note that the leaves of this tree represent the current KA-partition; when the algorithm terminates the leaves constitute the coarsest self-stable KA-partition.

The correctness of the naive algorithm relies on the following observations, which are adaptations of similar ones found in [PT87].

1. If $P' \trianglelefteq P$ and P is stable with respect to C', then so is P'.
2. If P is self-stable, then P is stable with respect to any P' such that $P \trianglelefteq P'$.
3. If $P \trianglelefteq P'$ then $split(C', P) \trianglelefteq split(C', P')$ for any C'.
4. *split* is commutative: $split(C_1, split(C_2, P)) = split(C_2, split(C_1, P))$.

To analyze the complexity of this procedure we first introduce the following notation. Let $\mathcal{S}$ refer to the set of bisimulation-equivalence classes of transition system $\langle S, E\rangle$; thus $|\mathcal{S}|$ represents the number of such equivalence classes.

The loop in the naive algorithm executes at most $|\mathcal{S}_1| \cdot |S_2|$ times, since each bisimulation-equivalence class has an auxiliary set that can only decrease $|S_2|$ times. Furthermore, each execution of the loop can be performed in $|E_1| + (|\mathcal{S}_1| \cdot |E_2|)$. The first term counts the total amount of time over all splitters for updating kernel sets in the KA-pairs, while the second reflects the time for updating the auxiliary sets. In addition, only the current KA-partition needs to be stored. This leads to the following.

Theorem 3. *The naive algorithm computes the relational coarsest KA-partition in $O(|\mathcal{S}_1| \cdot |S_2| \cdot (|E_1| + (|\mathcal{S}_1| \cdot |E_2|)))$ time and $O(|\mathcal{S}_1| \cdot |S_2|)$ space.*

4.2 An Improved Algorithm

The algorithm just given uses the "naive" partition-refinement strategy of [PT87] as a basis for computing the coarsest self-stable KA-partition; it also makes no attempt to exploit bisimulation equivalence in the "upper" transition system. These observations suggest two avenues for an improved routine.

1. Use the "three-way splitting" partition-refinement algorithm of [PT87].
2. Maintain equivalence classes of states in the auxiliary sets of KA-pairs.

This section shows how these ideas may be combined into a single efficient procedure. We begin by briefly reviewing the three-way splitting idea of [PT87].

Partition-refinement and three-way splitting. Paige and Tarjan [PT87] exploit Hopcroft's "process the smaller half" strategy for minimizing deterministic state machines [AHJ74] to give a more efficient algorithm for solving the RCPP. The main idea is to split a partition with respect to two splitters by only processing the transitions entering the smaller of the two. This approach may split equivalence classes into three pieces, and we thus refer to it as "three-way splitting."

The key insight behind three-way splitting is as follows. Let $T = \langle S, E \rangle$, and consider block B in partition P of S. Suppose B is stable with respect to a set (former block) $C \subseteq S$, and suppose further that C has been split into C_1 and C_2 and B must now be split with respect to these. If $B \subseteq E^{-1}(C)$, then splitting B with respect to both C_1 and C_2 yields (up to) three new equivalence classes.

$$\begin{aligned} B_{11} &= B \cap E^{-1}(C_1) \cap E^{-1}(C_2) \\ B_{12} &= (B \cap E^{-1}(C_1)) - E^{-1}(C_2) \\ B_2 &= (B - E^{-1}(C_1)) \cap E^{-1}(C_2) \end{aligned}$$

B_{11} contains states from B having transitions into both C_1 and C_2, B_{12} contains states from B having transititions into C_1 but not C_2, while B_2 contains states from B with no transitions into C_1 but transitions into C_2. Note that $B = B_{11} \cup B_{12} \cup B_2$.

Algorithmically, [PT87] gives a way to compute B_{11}, B_{12} and B_2 by scanning the smaller of C_1 and C_2. To achieve this one must know, for each state s in B, how many of s's transitions lead to states in C. One can then construct similar counts for each state in B and the smaller of C_1 and C_2 (call it C_{small}) by processing each transitions leading into C_{small}. That is,

$$\begin{aligned} B_{11} &= \{s \in B \mid 0 < |E(s) \cap C_{\mathsf{small}}| < |E(s) \cap C|\} \\ B_{12} &= \{s \in B \mid 0 = |E(s) \cap C_{\mathsf{small}}|\} \\ B_2 &= \{s \in B \mid |E(s) \cap C_{\mathsf{small}}| = |E(s) \cap C|\} \end{aligned}$$

To exploit this observation the three-way splitting algorithm maintains a list of *compound splitters*, which are trees of splitters with respect to whose roots the current partition is stable. In the previous example, C would be the root of a compound splitter, while C_1 and C_2 would be the children of C. When three-way splitting is done with respect to C_1 and C_2, C_1 and C_2 become the roots of new compound splitters if they have themselves been previously split. More details may be found in [PT87,Fer90].

Adapting three-way splitting to KA-partitions. To adapt three-way splitting to KA-partitions it is convenient to recall how our algorithms construct "partition trees" labeled by KA-pairs. As was the case in the previous algorithm, we maintain the following invariant in this tree: the right child of a two-child node has the same auxiliary set as its parent. The leaves of the tree constitute the current KA-partition, and a compound splitter is a subtree with the property that the current KA-partition is stable with respect to the label of the subtree's root.

Let $\langle B, X \rangle$ be a KA-pair in the current KA-partition, and let C be the root node of compound splitter having two subtrees. Assume further that the KA-pair labeling C is $\langle B', X' \rangle$ and that the label of C's left child, C_1, is $\langle B'_1, X'_1 \rangle$ and that the label of its right child, C_2, is $\langle B'_2, X' \rangle$. (Recall that the right child's auxiliary set is the same as its parent's.) Then the result of splitting $\langle B, X\rangle$ with respect to both C_1 and C_2 will in general be the following.

$$\begin{aligned}
\langle B_{11}, X_1\rangle &= \langle B \cap E_1^{-1}(B_1') \cap E_1^{-1}(B_2'), X \cap E_2^{-1}(X)\rangle \\
\langle B_{12}, X_1\rangle &= \langle (B \cap E_1^{-1}(B_1')) - E_1^{-1}(B_2'), X \cap E_2^{-1}(X)\rangle \\
\langle B_2, X\rangle &= \langle (B - E_1^{-1}(B_1')) \cap E_1^{-1}(B_2'), X\rangle
\end{aligned}$$

The characterizations of the kernel sets B_{11}, B_{12} and B_2 follows from the discussion of the Paige-Tarjan algorithm above, but the associated auxiliary sets deserve further comment. Regarding $\langle B_2, X\rangle$, recall that since C is a node in the partition tree whose right child is C_2, the auxiliary sets labeling C and C_2 are the same. Since $\langle B, X\rangle$ is stable with respect to C, it follows that every state in X has a transition into X', the auxiliary set of C and hence of C_2. Since B_2 consists of the states of B with no transitions into C_1, it follows that every state in X is a candidate for simulating every state in B_2.

On the other hand, states in B_{12} have transitions into C_2 but not C_1. Since the auxiliary set of C_1 is a subset of the auxiliary set of C, not every state in X, the auxiliary set of B, can safely simulate states in B_{12}: only those with a transition into X' $(X \cap E_2^{-1}(X'))$ can. A similar line of reasoning holds for $\langle B_{11}, X_2\rangle$. Note (suprisingly)that the auxiliary sets of B_{11} and B_{12} are the same.

Figure 1 shows the resulting tree structure rooted at B. Node $\langle B_1, X_1\rangle$ is inserted so that the partition tree is binary; implicitly, $B_1 = B_{11} \cup B_{12}$. Note that the invariant regarding right children is maintained.

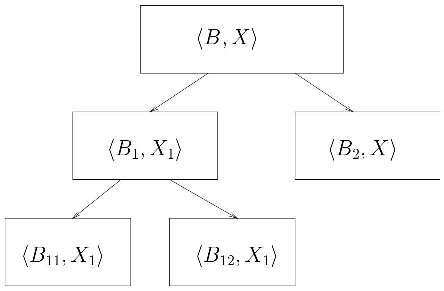

Fig. 1. Three-way splitting.

An additional subtlety in KA-partitions is that compound splitters can have one child rather than two. This arises when a node's auxiliary set is pruned without its kernel set being split. Such splitters can be treated as special cases of two-child splitters in which the right child's kernel set is empty. A KA-pair split by such a splitter will only have one child, as its kernel set cannot be split.

To implement these ideas efficiently we use several data structures. For each KA-pair $D = \langle B, X\rangle$ in the current partition (i.e. at the leaves of the partition tree) we use doubly-linked lists $D.B$ and $D.K$ to represent B and X, respectively.

This ensures constant-time insertions and deletions. Internal tree nodes do not have their kernel and auxiliary sets represented explicitly; rather, they may be reconstructed from the leaves that are descendants of the node. We also use the following data structure for efficiency reasons.

Kernel count table. To each compound splitter $C = \langle B', X' \rangle$ we associate a hash table $C.K$ that, for each state $s \in S_1$ in the "lower" transition system, maintains $|B' \cap E_1(s)|$ (i.e. the number of transitions from s into the kernel set of C). We use $C.K(s)$ to stand for the count associated with state $s \in S_1$. In three-way splitting, it suffices to compute $C_{\mathsf{small}}.K$, where C_{small} is the smaller child of C, in order to compute B_{11}, B_{12}, B_2 and $C_{\mathsf{big}}.K$, where C_{big} is the larger of C's children.

Auxiliary count table. In analogy with $C.K$, $C.A$ records, for each $s \in S_2$ in the "upper" transition system, the quantity $|C.X \cap E_2(s)|$. So $C.A(s)$ is the number of transitions s has into the set stored in $C.X$.

Incoming node lists. For each potential compound splitter $C = \langle B', X' \rangle$, $C.F$ records the list of KA-pairs whose kernel states have transitions to B'. This information is needed to ensure that auxiliary sets are refined properly when nodes are split with respect to C. In particular, if C's right child, C_2, has a smaller kernel set than its sibling C_1, and KA-pair $D = \langle B, X \rangle$ is such that B only has transitions into C_1, then the auxiliary set of D's (only) child, which would be $X \cap E_2^{-1}(X')$, will not be computed if only blocks with transitions into C_2 are analyzed.

For leaf nodes C, $C.X$ stores the auxiliary set associated with the node. For internal nodes D, in contrast, we use $D.X$ to store *difference sets*. More specifically, rather than storing the entire auxiliary set of the KA-pair $\langle B, X \rangle$ associated with D in $D.X$, we store only those elements of the set that are not in the auxiliary set of D's left child. Let D_1 be the left child of D, and let $\langle B_1, X_1 \rangle$ be D_1's KA-pair. Then the set of states stored in $D.X$ is $X - X_1$. When doing three-way splitting on KA-pair $\langle B, X \rangle$ with respect to compound splitter $C = \langle B', X' \rangle$ whose left child C_1 is labeled $\langle B'_1, X'_1 \rangle$, $C.X$ can be used to compute the auxiliary set X_1 of $\langle B_{11}, X_1 \rangle$ using the following identity.

$$X \cap E_2^{-1}(X') = X - \{s \in X \mid E_2(s) \subseteq (X' - X'_1)\} \qquad (1)$$

The use of difference sets has efficiency ramifications; in particular, the amortized analysis of the complexity of the algorithm relies on the use of difference sets.

Special care must be taken for partition-tree nodes having only one child. Since only leaves store KA-pairs, calculating the kernel set of an internal node D requires gathering all the kernel sets of the leaves in D's subtree. In the Paige-Tarjan algorithm [PT87], this may be done in time proportional to the size of D's kernel set, since every internal node has two children and kernel sets are disjoint. Because of the existence of single-child chains in our tree, this does not immediately apply. To solve this problem, we use *path compression*: we add a field $D.root$ that points to the first node on a single-child chain that D may be part of. For nodes that are the roots of such chains, we add an additional field, $D.end$, that points to the end of its single-child chain.

Bisimulation equivalence in auxiliary sets. The second direction for improving the algorithm involves the exploitation of bisimulation equivalence classes in auxiliary sets. The basic approach is to maintain a current partition for states in the "upper" transition system, $T_2 = \langle S_2, E_2 \rangle$. Each block represents an approximation to the bisimulation equivalence classes of T_2. The fields $D.X$ then point to lists of these equivalence classes rather than to states.

More specifically, we use *auxiliary list tables (ALTs)* to store auxiliary sets. An ALT has two kinds of entries.

Base entries point to lists of states in the upper transition system. Taken together, the base entries form a partition of the state space.

List entries point to lists of base entries. These entries will in turn be pointed to by the auxiliary set components $C.X$ of a node C in the partition tree.

The lists in ALT are implemented as doubly-linked lists in order to support $O(1)$ insertions and deletions. In addition, for $s \in S_2$, $baseOf(s)$ retrieves the base entry s belongs to; this can be implemented in constant time by maintaining an array storing each auxiliary state's base entry b and position in the state list of b. We also associate with each base entry b a field $b.t$, which is used for splitting b, and a hash table $b.R$ indexed by the list entries it belongs to; $b.R$ stores the positions of b in these list entries so that b can be quickly deleted. Together $baseOf$ and $b.R$ allow the query $s \in l?$, where l is a list entry, to be answered in $O(1)$ time: first look up the base entry b that s belongs to, then look in $b.R$ to see if there is an entry for l. $mkListEntry(l)$ is an initialization function; it creates a list entry for a set of states by first creating a new base entry for this set and then a new list entry containing only this base entry. $addBaseToList(b, l)$ adds a base entry b to a list entry l and saves b's position in the doubly-linked list of l to $b.R$. $removeBaseFromList(b, l)$ removes b from l and adjusts $b.R$ accordingly. $duplicate(l)$ returns a new list entry whose doubly-linked list contains the same bases as l.

During the execution of the algorithm base entries will periodically require splitting, since states in the same base entry may be determined not to be bisimulation equivalent. For example, this happens when auxiliary sets are "pruned" during three-way splitting: some states in a base entry b may be determined to have transitions to a given auxiliary set (which may be shown always to be a union of bisimulation equivalence classes in the upper system), while others do not. In this case the former states are moved to $b.t$. Then operation $processSplitBases$ splits bases whose $b.t$ list is non-empty; such base entries are called *split bases*. For a given split base b, $processSplitBases$ creates a new base entry b' for the states in $b.t$ if b itself is non-empty and moves $b.t$ to b'. The procedure then updates list entries appropriately: it takes another parameter, a list of pairs of list entries, with the first list of each pair representing an "old home" of b and the second representing the "new home" for b'. (In general, the former list will be the auxiliary list of a node and the later an auxiliary list of a left child. The former should be turned into a difference list, while the latter is expecting to be populated with base entries.) No pair shares the same "old list" component, so this list of pairs can be organized as hash table, enabling membership queries to be done in $O(1)$ time. For all other list entries that are not in an "old list", the routine adds the new base entry b' to those already containing b so that the states they contain remain unchanged.

The algorithm in detail. Our algorithm computes the Relational Coarsest KA-Partition from $T_1 = \langle S_1, E_1 \rangle$ to $T_2 = \langle S_2, E_2 \rangle$ in several stages. It starts by building a KA-partition containing one KA-pair, $\langle S_1, S_2 \rangle$: every state in S_2 is assumed to simulate every state in S_1, and all states in S_1 and S_2 are assumed to be bisimulation equivalent.

The first step in the algorithm is to stabilize the KA-partition with respect to the single KA-pair $\langle S_1, S_2 \rangle$. After creating a node C and initializing $C.B$ to S_1 and $C.X$ to S_2, $C.B$ is split into states having transitions and those that do not; the former are assigned to $C_1.B$, where C_1 is the left child of C, while the latter are assigned to $C_2.B$, the right child of C. The counters in $C.K$ are also initialized to the number of transitions each state has (except that states without transitions are not touched). Then the auxiliary list of C is copied into $C_2.X$, and $C_1.X$ computed by scanning the transitions leading into $C.X$. This procedure may also induce a split in the base entry containing S_2, since states without transitions cannot be bisimilar to those that do. The latter states are assigned to a base entry that becomes part of $C_1.X$, while those that do not become the elements of $C.X$, which is now a difference list. At the end, there is a single compound splitter, C, with left child C_1 and right child C_2.

The algorithm then loops, repeatedly removing splitters from a list of splitters, performing the split, and potentially adding new splitters, until the list of splitters is empty. Given a (compound) splitter C, the kernel sets of the current partition (leaves in the partition tree) are split by processing the child containing the smaller splitter. This entails decrementing $C.K(s)$ and incrementing $C_{\mathsf{small}}.K(s)$, where C_{small} is the smaller child of C. Then each KA-pair D that is touched in this process is examined and split using three-way splitting. Temporary fields $D.B_0$ and $D.B_1$ are used for this purpose.

Following the three-way splitting operations, the auxiliary sets for KA-pairs with transitions into C are created. Right children are given copies of the auxiliary sets of their parents by copying list entries in the ALT, and base entries are split when some states are determined to have transitions into some sets that others cannot match. Finally, splitter lists are updated; C_1 and C_2 are added as compound splitters if they have children, as well as other nodes that were split and yet were not part of any splitter.

Theorem 4. *The algorithm converges, and the leaf KA-pairs form the relational coarsest KA-partition when the algorithm terminates.*

The next theorems characterize the complexity of our algorithm. Recall that for transition system $T = \langle S, E \rangle$, $\mathcal{S}$ refers to the set of bisimulation equivalence classes T. We also use $\mathcal{E}$ to refer to the transition relation on $\mathcal{S}$ defined by: $\langle h, h' \rangle \in \mathcal{E}$ if and only if there exist $s_1 \in h_1, s_2 \in h_2$ such that $E(s_1, s_2)$.

Theorem 5. *The overall running time of the three-way splitting algorithm with path compression is* $O(|\mathcal{S}_1| \cdot |\mathcal{S}_2| + |E_1| \cdot log(|S_1|) + |\mathcal{S}_1| \cdot |E_2| + |\mathcal{E}_1| \cdot |\mathcal{S}_2|)$.

Theorem 6. *The space required by the three-way splitting algorithm is bounded by* $O(|T_1| + |T_2| + |\mathcal{S}_1| \cdot |\mathcal{S}_2|)$.

We conclude this section by comparing our time and space efficiency with the simulation algorithm in [BP95]. That procedure ran in $O(|S_1| \cdot |S_2| + |S_1| \cdot |E_2| + |S_2| \cdot |E_1|)$ time and $O(|T_1| + |T_2| + |S_1| \cdot |S_2|)$ space. Our complexity results replace many occurences of E_i and S_i with $\mathcal{E}_i$ and $\mathcal{S}_i$; indeed, the only worst-case penalty our procedure pays is the $|E_1| \cdot log(|S_1|)$ factor, which is due to the three-way splitting our procedure performs. The experimental results in the next section nevertheless indicate that our procedure works better in practice.

5 Experimental results

To assess the practical performance of our algorithm we implemented it in the Concurrency Workbench of the New Century (CWB-NC), a verification tool for finite-state systems (see `www.cs.sunysb.edu/~cwb` to obtain the system). The CWB-NC analysis routines work on labeled transition systems, so we adapted our algorithm to this setting by adding an action parameter to the splitting operation and then splitting a partition with respect to all actions, given a splitter. The approach followed is similar to that presented in [Fer90] for adapting the Paige-Tarjan algorithm [PT87] to labeled transition systems. The implementation of our algorithm involves 2,045 lines of Standard ML of New Jersey, with approximately a quarter of this total being devoted to maintaining ALT tables.

We then tested four different simulation algorithms on case studies included in the CWB-NC release. The four simulation algorithms checked included: the implementation of the Bloom-Paige algorithm [BP95,CC95] included in the CWB-NC release; our naive algorithm; our algorithm with the ALT data structure but without path compression; and our full algorithm. In all cases *early termination* is used: when two start states are found to be unrelated, the algorithm terminates. We ran the implementations on two different classes of systems.

Railway-signaling schemes. Three models of the British Rail Slow-Scan communications protocol as modeled in [CLNS96] were compared to each other. The systems are implemented in a version of CCS with priorities.

Alternating-bit protocols. Different versions of the alternating-bit protocol were compared, including ones that deadlocked and chains of cells.

All testing was done on a Sun Ultra SparcIIi with two 336 MHz processors and 3 GB of main memory. All times are reported in seconds of CPU time.

The results for the railway models are reported in Table 2, while those for the alternating bit protocol may be found in Table 4. The columns headed "Bloom-Paige" present times for the Bloom-Paige algorithm, "Naive KA-part" for our naive algorithm, "Sim-ALT" for our more sophisticated algorithm without path compression, and "Sim-ALT-PC" for our algorithm with path compression. We also compared the performance of Sim-ALT and the Bloom-Paige algorithm when the systems are first minimized with respect to strong bisimulation. Tables 1 and 3 give the sizes of the systems before and after minimization. In all case "# states / # trans" refers to the number of reachable states and transitions.

Table 1. Railway system sizes before and after minimization.

	# states	# trans	# bisim classes	# of bisim trans
basicSS	312	801	287	713
recoverySS	1100	2801	789	2233
ftolerantSS	11905	33760	7485	26165

Table 2. Railway simulation results.

Agent 1 Agent 2	ans	Bloom-Paige	min + Bloom-Paige	Naive KA-Part	Sim-ALT	Sim-ALT -PC	min + Sim-ALT
basicSS recoverySS	T	124.74	7.14+ 62.91	24.22	7.52	9.62	7.14+ 2.44
basicSS ftolerantSS	F	N/A^1	131.48+ 2109.90	330.78	139.80	139.02	131.48+ 26.63
recoverySS ftolerantSS	F	N/A^1	137.21 + N/A^1	634.10	278.05	273.99	137.21 + 32.15
recoverySS basicSS	F	186.23	7.14 + 157.28	28.67	14.95	13.60	7.14 + 1.39
ftolerantSS basicSS	F	10831.63	131.48 + 1724.00	284.85	192.50	194.12	131.48 + 23.99
ftolerantSS recoverySS	F	N/A^1	137.21 + 31104.24	192.95	256.73	267.59	137.21 + 17.91

1. Memory allocation error after $>$ 4 hour

Based on the times presented one may draw the following conclusions.

1. *Our algorithms dramatically outperform the Bloom-Paige algorithm in time and space.* Even the naive algorithm substantially outperforms Bloom-Paige. The degrees of improvement are often quite startling, ranging up to a factor of 100 and beyond; we believe this is due to the space efficiency of our algorithms, which causes them to use less virtual memory.
2. *When there are few equivalence classes, minimizing and then running Bloom-Paige can be competitive with our algorithms running on unminimized systems.* The ABP results suggest this in particular: minimization can dramat-

Table 3. ABP system sizes before and afterminimization.

	# states	# trans	# bisim classes	# bisim trans
ABP-lossy	57	130	15	32
ABP-safe	49	74	18	32
Two-link-netw	1589	6819	197	791
Three-link-netw	44431	280456	2745	16188
Two-link-netw-safe	1171	3153	196	662

Table 4. ABP simulation results.

Agent 1 Agent 2	ans	Bloom- Paige	min+ Bloom-Paige	Naive KA-Part	Sim-ALT	Sim-ALT -PC	min+ Sim-ALT
ABP-lossy Two-link-netw	F	11.07	2.00+ 0.20	4.21	2.78	5.57	2.00+ 0.12
ABP-lossy Three-link-netw	F	7104.53	89.91+ 14.60	185.00	443.48	476.82	89.91+ 2.66
Two-link-netw Three-link-netw	F	N/A	91.81+ 232.10	4320.63	662.97	625.84	91.81+ 6.71
Two-link-netw ABP-lossy	F	18.34	2.00+ 0.24	5.52	6.31	5.30	2.00+ 0.13
Three-link-netw ABP-lossy	F	1116.19	89.91+ 5.79	231.63	137.63	135.59	89.91+ 2.31
Three-link-netw Two-link-netw	F	*N/A*[1]	91.81+ 160.25	4966.73	473.28	561.37	91.81+ 88.47
ABP-safe ABP-lossy	T	0.16	0.08 + 0.02	0.20	0.09	0.11	0.08+ 0.02
Two-link-netw-safe Two-link-netw	T	432.58	1.99 + 3.82	84.27	5.60	5.68	1.99 + 0.95

1. Memory allocation error after > 1 hour

ically improve the performance of Bloom-Paige. This also leads us to believe that paging is a major source of inefficiency in Bloom-Paige.

3. *Sim-ALT substantially outperforms Bloom-Paige when both are run on minimized systems.* This result may seem surprising, given that our algorithm is intended to combine the benefits of minimization with those of simulation checking. Howevever, as Theorem 5 shows, our algorithm's time complexity still contains factors involving the number of transitions in the input systems.
4. *Path compression is a net loss for our algorithm.* In order to obtain the complexity result in Theorem 5 it was necessary to introduce path compression. However, the performance figures suggest that this improvement does not materialize in practice.

6 Conclusions and Future Work

This paper has presented an algorithm for determining whether or not one transition system can simulate another. The procedure combines ideas from traditional simulation algorithms with notions found in bisimulation-equivalence procedures; the resulting routine has asymptotic time- and space-complexities that approach those of the best-known algorithm [BP95,CC95]. In practice, our approach dramatically outperforms the existing routines, owing to the fact that our procedure exploits bisimulation equivalence to reduce both time and space consumption.

As future work we plan to investigate the use of our ideas to improve mu-calculus model checking. It is known that bisimilar systems satisfy the same mu-calculus formulas; consequently, combining a bisimulation-equivalence algorithm with a model checker could also yield potentially dramatic performance improvements in practice. We also wish to investigate adaptations of our algorithm in the computation of other relations, including the so-called "weak" simulation ordering in which transitions labeled by internal actions are allowed to be "absorbed" into transitions labeled by external actions. It should also be noted that our algorithm is *global*: the transition system must be built before the routine may be run. It would be interesting to investigate combining our ideas with on-the-fly approaches to system minimization in order to avoid the *a priori* construction of the system state spaces [BFH+92].

Related work. Bloom [Blo89] proposed an algorithm for *ready simulation* that runs in $\theta((|E_1|+|E_2|)\cdot(|S_1|+|S_2|)^6)$ time. Bloom and Paige improved this result to $O(|S_1|\cdot|T_2|+|S_2|\cdot|T_1|)$ in [BP95]; similar ideas may also be found in [CS90], where preorder-checking is reduced to model checking, and in [CC95,HHK95].

References

[AHJ74] A.Aho, J. Hopcroft, and J.Ullman. *Design and Analysis of Algorithms.* Addisom Wesley, 1974.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property-preserving simulations. In G.v. Bochmann and D.K. Probst, editors, *Computer Aided Verification (CAV '92)*, volume 663 of *Lecture Notes in Computer Science*, pages 260–273, Montréal, June/July 1992. Springer-Verlag.

[BFH+92] A. Bouajjani, J.C. Fernandez, N. Halbwachs, C. Ratel, and P. Raymond. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–271, June 1992.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[Blo89] B. Bloom. *Ready Simulaton, Bisimulation, and the Semantics of CCS-Like languages.* PhD thesis, Massachusetts Institute of Technology, Aug. 1989.

[BP95] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, June 1995.

[CC95] U. Celikkan and R. Cleaveland. Generating diagnostic information for behavioral preordering. *Distributed Computing*, 9:61–75, 1995.

[Cel95] U. Celikkan. *Semantic Preorders in the Automated Verification of Concurrent Systems*. PhD thesis, North Carolina State University, Raleigh, 1995.

[CH93] R. Cleaveland and M. C. B. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[CLNS96] R. Cleaveland, G. Luettgen, V. Natarajan, and S. Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, 17:50–62, 1996.

[CS90] R. J. Cleaveland and B. Steffen. When is 'partial' adequate? a logic-based proof technique using partial specifications. In *Proceedings of 5th Annual IEEE Symposium on Logic in Computer Science*, Philadelphia, PA, June 1990.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[DNH83] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1983.

[Fer90] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Sicence of Computer Programming*, 13:219–236, 1989/90.

[HHK95] M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In 36^{th} *Annual IEEE Syposium on Foundations of Computer Science*, pages 453–462. Computer Society Press, 1995.

[Jon91] B. Jonsson. Simulations between specifications of distributed systems. In J.C.M. Baeten and J.F. Groote, editors, *CONCUR '91*, volume 527 of *Lecture Notes in Computer Science*, pages 346–360, Amsterdam, August 1991. Springer-Verlag.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations—part i: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*. BCS, 1971.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.