Bisimulation and Model Checking

Kathi Fisler and Moshe Y. Vardi*
Department of Computer Science
Rice University
6100 S. Main, MS 132
Houston, TX 77005-1892
{kfisler, vardi}@cs.rice.edu
http://www.cs.rice.edu/{~kfisler, ~vardi}

June 15, 1999

Abstract

State space minimization techniques are crucial for combating state explosion. A variety of verification tools use bisimulation minimization to check equivalence between systems, to minimize components before composition, or to reduce a state space prior to model checking. This paper explores the third use in the context of verifying invariant properties. We consider three bisimulation minimization algorithms. From each, we produce an on-the-fly model checker for invariant properties and compare this model checker to a conventional one based on backwards reachability. Our comparisons, both theoretical and experimental, lead us to conclude that bisimulation minimization does not appear to be viable in the context of invariance verification, because performing the minimization requires as many, if not more, computational resources as model checking the unminimized system through backwards reachability.

Keywords: Bisimulation minimization, model checking, invariant properties, on-the-fly model checking

1 Introduction

State-of-the-art model checkers are enjoying substantial and growing use on real-world problems. Industrial verification groups apply this technology to projects ranging from floating-point arithmetic units [12] to large asynchronous speed-independent circuits [20]. Despite the advances, however, the growing size of current and future semiconductor designs seriously challenges current model checking technology. In particular, the unremitting increase in design complexity, which manifests itself as the state-explosion problem, remains a serious obstacle to industrial-scale verification.

Various techniques reduce the size of the state space that a model checker must analyze. Some decompose designs into smaller components which are analyzed separately; combining results on the smaller components yields results on the full design [23, 29]. Others reduce the size of individual components through some form of abstraction [11, 18]. An abstraction hides some information from a state space to yield a smaller state space. Ideally, operations over the smaller state space should use less resources than over the original state space. Towards this end, abstractions are often applied as a pre-processing phase to model checking [19]. To be useful in practice, however, abstractions must preserve the properties that a designer wishes to verify. The choice of a suitable abstraction technique therefore depends on the properties of interest.

Bisimulation minimization [31] provides an abstraction technique that preserves the truth and falsehood of all μ-calculus (and hence all CTL*, CTL, and LTL) properties [25]. This technique is particularly appealing in the context of symbolic model checking [3, 13] for two reasons. First, bisimulation can be computed as the fixpoint of a simple boolean expression, so it is easily expressed symbolically. Second, unlike many other abstraction techniques, it can be computed automatically, which is consistent with the automated spirit of model checking. In earlier work,

*Supported in part by NSF grants CDA-9625898, CCR-9628400 and CCR-9700061, and by a grant from the Intel Corporation.

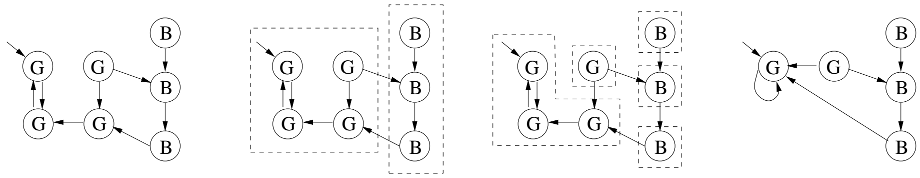


Figure 1: An example of bisimulation minimization on a state space with respect to one atomic proposition. The states satisfying the proposition are labeled G; the other states are labeled B. The original transition system appears on the far left. The dashed boxes in the second diagram show the initial partition. The boxes in the third diagram show the final equivalence classes. The minimized system appears on the far right.

we showed experimentally that using bisimulation minimization as a pre-processing phase to model checking does reduce the resource requirements of model checking [22]. However, our work also showed that the cost of performing bisimulation minimization often significantly exceeds that of model checking. It is therefore unclear that performing bisimulation minimization before model checking saves resources over simply model checking the original system.

Integrating minimization and model checking, however, might provide the best of both approaches. Minimization and model checking perform similar operations on a state space, so merging them into a single pass should avoid re-duplicating work. Furthermore, it would halt an otherwise expensive minimization once the property of interest was determined to fail. This paper considers this approach in the context of verifying invariances, which is the most fundamental model checking task. We consider three bisimulation minimization algorithms. From each, we produce a novel BDD-based on-the-fly model checker for invariances. We then characterize the sets of states that it computes at each iteration and compare those to the sets of states computed during model checking. We show a strong correlation between these sets, which suggests not only that the integrated algorithms are less efficient than model checking, but also that the original minimization algorithms are less efficient than model checking.

We also provide analytical and experimental comparisons. Although the various algorithms compute similar sets of states, they do so in very different ways; the relative behavior of the BDD operations across these different methods is therefore unpredictable. To account for this, we calculate lower bounds on the numbers of symbolic computations of various types (such as image computations and intersections) used in each algorithm and compare the number of iterations needed to check invariances in each approach. This combination of analytical and empirical evidence strongly suggests that performing bisimulation minimization on the entire design does not improve on the resource usage of symbolic model checking.

Section 2 provides an overview of bisimulation minimization. Necessary terminology appears in Section 3. Section 4 reviews model checking for invariant properties. Section 5 compares three minimization algorithms to model checking. Experimental results appear in Section 6. Sections 7 and 8 discuss related work and our conclusions, respectively. An appendix contains the proofs of certain intermediate lemmas.

2 Bisimulation Minimization

Bisimulation minimization algorithms partition a state space into equivalence classes such that states in the same class are observationally equivalent with respect to the system's behavior. In the context of model checking, we are often interested in how the system behaves with respect to only some of its variables. We can therefore compute bisimulation relative to a given subset of a system's atomic propositions. For example, to verify a particular property, we minimize with respect to the atomic propositions that it contains. For an invariance, we can minimize with respect to one (possibly new) atomic proposition that is true in exactly those states that satisfy it.

The states in each equivalence class under bisimulation agree on both the values of the atomic propositions of interest and on their next-state transitions to other classes. Like most bisimulation minimization algorithms, the ones discussed in this paper follow a common outline, as shown in Figure 1. First, they group states into classes based on the atomic propositions. Next, they repeatedly split existing classes into new ones until all states in a class agree on their next-state transitions to other classes. For example, the bottom-most B state in Figure 1 splits off from the other two B states because it can reach a G state, while the others cannot. The algorithms stop when no more classes need to be split. The minimized system contains one state from each remaining class (a *representative*), with all edges to a class redirected to that state.

The naïve algorithm for computing bisimulation is defined relative to a transition system $\langle S, R, AP, L, init\rangle$, where S is a set of states, R is a total transition relation over S, AP is a set of atomic propositions, L is a mapping from AP to subsets of S, and $init \in S$ is a single initial state. Computing the least fixpoint of the following expression yields the complement of the maximum bisimulation relation B:

$$\begin{gathered} \exists\, ap \in AP'\ \neg(L(s_1, ap) \Leftrightarrow L(s_2, ap)) \vee \\ \exists\, s_1'\ R(s_1, s_1') \wedge \neg(\exists s_2'(R(s_2, s_2') \wedge B(s_1', s_2'))) \vee \\ \exists\, s_2'\ R(s_2, s_2') \wedge \neg(\exists s_1'(R(s_1, s_1') \wedge B(s_1', s_2'))) \end{gathered}$$

The first line partitions states based on the atomic propositions. The last two lines separate states that do not agree on their next-state transitions.

This algorithm has two shortcomings in the context of symbolic model checking. First, it computes the *relation*, rather than the individual equivalence classes. The BDD for the relation requires twice as many variables as the BDDs for the classes. This can lead to BDD explosion even on small examples. Experimental work confirms that the BDDs used to compute the bisimulation relation do get overly large in practice [10, 22], which suggests that computing bisimulation relations is not a feasible approach to algorithmic state-space reduction. The algorithms considered in this paper compute the equivalence classes instead of the relation. As we have argued in previous work [22], this makes a significant difference in practice and makes bisimulation minimization feasible. The naïve algorithm's other shortcoming is its failure to distinguish between reachable and unreachable states. In Figure 1, for example, no B state is reachable, yet the algorithm splits the unreachable class of B states into sub-classes. In the context of model checking, this work is unnecessary. The algorithms considered in the paper take various approaches to addressing this issue, as discussed in Section 5.

From our previous experimental work with one of these algorithms (Lee-Yannakakis, [30]), we know that bisimulation minimization can yield substantial reductions in the size of the reachable state space for examples arising in model checking. Furthermore, these reductions often yield noticeable reductions in the time needed to analyze the state space during model checking. Table 1 provides experimental data for these observations over a range of examples. Comparing the time needed to model check the original design with the combined times for minimization and model checking the minimized design, minimization does not appear to be worthwhile. This observation motivated the work described in this paper.

3 Terminology

We use Figure 1 to define terminology for the rest of the paper. A *block* is a set of states, all of which agree on whether the given invariant property holds. A block may designate one of its states as its *representative*. A *partition* is a set of disjoint blocks that cover the state space. Each algorithm starts with an *initial partition* into two blocks: the *good block* consists of those states that satisfy the given invariant property, and the *bad block* consists of those states that fail to satisfy it (the *bad* states). Partition P_1 *refines* partition P_2 iff every block in P_1 is contained in some block of P_2. A state s is *reachable* iff the pair $(init, s)$ is in the transitive closure of the transition relation; this indicates a path from the initial state to s. A block is reachable if it contains a reachable state; in Figure 1, none of the B blocks are reachable. Reachable blocks may also contain unreachable states; in Figure 1, the block of three G states contains the unreachable rightmost G state. At each iteration through each minimization algorithm, some reachable block contains the initial state. We call this block the *initial block*.

A block B_1 is *stable* with respect to block B_2 iff either all states in B_1 have transitions to states in B_2 or no state in B_1 has a transition to a state in B_2. If B_1 is not stable with respect to B_2, B_2 is called a *splitter* of B_1. In the second diagram in Figure 1, the block of G states is a splitter for the block of B states. A block is stable with respect to a partition iff it is stable with respect to each block in the partition.

We also use some standard notation on transition systems $\langle S, R, AP, L, init\rangle$. Given a subset S' of S, $pre_R(S')$ is the pre-image of S' under R and $post_R(S')$ is the image of S' under R. Since R is clear from context, we write simply pre and $post$. The complement of S' is denoted $\overline{S'}$.

Finally, we want to compare the algorithms with respect to how many operations of various types they require. Towards this end, we derive a lower bound for each algorithm that captures how many operations of each type it performs in the best case. We state these bounds in terms of the following variables: n, the number of iterations

	Reachable States	Reachable Blocks	MC Time	MC Memory	Minimization Time	Minimization Memory	Min. Design MC Time
arbiter (L)	73	64	0.6	2.98	11.7	3.62	0.2
eisenberg1* (S)	1611	289	3.1	3.87	58.5	4.8	1.7
eisenberg2 (L)	1611	1078	3.1	3.85	158.6	4.7	5.6
bakery1* (L)	2886	2546	8.4	3.64	354.3	5.6	17.3
bakery2 (L)	2886	2546	8.4	3.62	347.1	5.6	7.0
coherence (S)	94738	1	45.5	6.24	85	22	9.3
dcnew1 (S)	186876	1	1.8	4.12	0.9	4.03	0
dcnew2* (L)	186876	230	1.8	4.12	67.5	5.94	0.7
k_elev1 (L)	262	132	0.5	2.92	5.2	3.17	0.2
treearbiter4-1 (S)	5568	1	2.1	3.7	17.4	5.16	0
treearbiter4-2 (L)	5568	1344	2.1	4.03	485.7	6.28	1.5
elevator23 (S)	674077	1	8.0	8.24	4.3	8.52	1.4
elevator33 (S)	$1.51 * 10^8$	1	259.4	28	431.9	17	167.3
elevator43 (S)	$1.18 * 10^{10}$	1	1278.7	51	1624.0	42	871.5
fpmpy1 (S)	$7.15 * 10^{12}$	5	41.9	11	113.6	18	0.1
fpmpy2* (S)	$7.15 * 10^{12}$	15	52.2	16	375.6	18	2.2
fpmpy3 (S)	$7.15 * 10^{12}$	15	42.9	12	343.0	17	0.1
fpmpy4 (S)	$7.15 * 10^{12}$	15	42.7	11	440.1	26	0.2
fpmpy5* (S)	$7.15 * 10^{12}$	17	46.8	1	259.7	22	1.1
fpmpy6 (S)	$7.15 * 10^{12}$	15	44.3	12	308.6	22	0.2
minmax (L)	$2.06 * 10^{26}$	2	22.8	21	7.5	11	2.2
tcp1* (S)	$3.88 * 10^{22}$	32	2.7	7.08	48.3	9.7	1.2
tcp2 (S)	$3.88 * 10^{22}$	1	2.7	7.08	3.5	8.03	1.0

Table 1: Experimental data on the effect of bisimulation minimization on model checking. Designs have been minimized relative to particular properties. A * after an experiment's name indicates that the corresponding property fails. The S and L designations after the property name denote safety or liveness properties, respectively. In order to minimize the number of atomic propositions used in each experiment, the minimization routine uses one atomic proposition for each maximal subexpression that does not contain a temporal operator. The Reachable States column shows the number of reachable states in the original design. Reachable Blocks is the number of equivalence classes in the minimized system. The MC Time and Memory columns indicate the resource requirements for model checking the original design (in seconds and megabytes, respectively). The Minimization Time and Memory columns indicate the resource requirements for minimization using the Lee and Yannakakis algorithm [30]. The Min. Design MC Time column shows the time needed to model check the property on the minimized design. We were unable to isolate the memory required to model check the minimized system, due to constraints in our experimental framework.

$$
\begin{aligned}
F_0 &= Bad & (1)\\
F_{i+1} &= pre(F_i) - S_i & (2)\\
\\
S_0 &= Bad & (3)\\
S_{i+1} &= pre(F_i) \cup S_i & (4)
\end{aligned}
$$

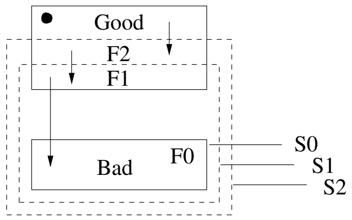


Figure 2: The equations for BR and how they carve up the state space. The dashed lines show both how BR divides the good block into the frontier sets, and which states lie in each set of explored states. The arrows indicate that each state in the source set points to some state in the target set. The circle represents the initial state; this example assumes the initial state is a good state at least three steps away from a bad state. Note that $F_{i+1} = S_{i+1} - S_i$.

through the main processing loop of the algorithm; M, the number of image operations; I, the number of intersection operations; U, the number of union operations; D, the number of set-difference operations; and E, the number of equality checks. These bounds will allow us to compare symbolic algorithms based on their worst-case behavior for each of these operations. A BDD-based comparison, for example, would render E constant, M exponential, and the remaining variables polynomial in the size of the sets being manipulated.

4 Model Checking Invariances

An invariance is true of a system iff it holds in all reachable states. Checking whether a system satisfies an invariance proceeds in one of two ways [26]. *Forwards reachability* starts from the initial state and follows the transition relation forwards, looking for a reachable bad state. *Backwards reachability* starts from the bad states and follows the transition relation backwards, looking for the initial state. Each approach terminates when the desired type of state is found or when the set of explored states hits a fixpoint. These algorithms carve the state space into disjoint sets of states, where the states in each set agree on their minimum distance from the initial state (forwards reachability) or to a bad state (backwards reachability). On each iteration, these algorithms split the set of unexplored states into those that reach (backwards reachability) or are reached from (forwards reachability) the explored states. Thus, reachability and minimization have a similar flavor. Since bisimulation minimization relies on pre-image, rather than image, computations, it appears more closely related to backwards reachability (henceforth, BR). This paper therefore compares BR to bisimulation minimization.

Formally, BR iterates over two sets of states, as shown in the equations in Figure 2. The *frontier states* (F) are the new states discovered on each iteration. The *explored states* (S) consists of all states that have been on the frontier during some previous iteration. The algorithm terminates when either $F_i = \emptyset$ or $init \in F_i$. Figure 2 also depicts how BR carves up the state space relative to these sets. Let n denote the number of iterations required for a given run of BR to terminate. If there is a path from the initial state to a bad state, n is the length of the shortest such path. If there is no such path, n is the length of the longest acyclic path from a good (but unreachable) state to a bad state. It is clear from the equations that each iteration requires one image computation, one union computation, and one difference computation. The termination checks at each iteration require one intersection (in the membership test) and two equivalence checks.[1] BR's computation lower bound is therefore $n * (M + U + D + 2E + I)$.

We finish this section with two lemmas about the sets S_i and F_i that we need for later proofs.

Lemma 1

$$S_i = F_0 \cup \bigcup_{j=1}^{i-1} pre(F_j)$$

[1] We test membership via intersection and an equivalence check, rather than use a direct membership test, because we have encountered problems with the membership function in the release of the BDD package used for this project.

Proof The proof is by induction on i. In the base case, i is zero. $S_0 = Bad = F_0$ by definition, so the base case holds. Assume $S_i = F_0 \cup \bigcup_{j=1}^{i-1} pre(F_j)$. By definition, $S_{i+1} = S_i \cup pre(F_i)$. Substituting for S_i via the inductive hypothesis,

$$\begin{aligned} S_{i+1} &= F_0 \cup \bigcup_{j=1}^{i-1} pre(F_j) \cup pre(F_i) \\ &= F_0 \cup \bigcup_{j=1}^{i} pre(F_j), \end{aligned}$$

which yields the desired result. □

Lemma 2 $pre(F_0 \cup \ldots \cup F_{i-1}) \subseteq F_0 \cup \ldots \cup F_i$.

Proof Expanding out the definitions of each F_j,

$$F_0 \cup \ldots \cup F_i = F_0 \cup (pre(F_0) - S_0) \cup \ldots \cup (pre(F_{i-1}) - S_{i-1}).$$

Applying Lemma 1 several times yields $F_0 \cup pre(F_0) \cup \ldots \cup pre(F_{i-1})$. Distributing pre over union yields the desired result: $pre(F_0 \cup \ldots \cup Fi - 1) \subseteq F_0 \cup \ldots \cup F_i$. □

5 Three Bisimulation Minimization Algorithms

As shown in Section 2, bisimulation minimization algorithms repeatedly locate and stabilize unstable blocks. This section discusses three bisimulation minimization algorithms: by Paige and Tarjan (henceforth PT) [32], Bouajjani, Fernandez, and Halbwachs (henceforth BFH) [5], and Lee and Yannakakis (henceforth LY) [30]. We chose these algorithms for the following reasons:

- **PT:** Has the best provable worst-case running time of traditional bisimulation minimization algorithms (those that stabilize both reachable and unreachable blocks).
- **BFH:** Improves on PT by choosing only reachable blocks to stabilize on each iteration; however, it may stabilize an unreachable block that was split off from the reachable block being stabilized in the current iteration.
- **LY:** Improves on BFH by never stabilizing an unreachable block.

By stabilizing few, if any, unreachable blocks, the BFH and LY algorithms are tailored to verification contexts. The PT algorithm, although not so tailored, is interesting because its stabilization loop chooses splitters instead of blocks to split (LY and BFH do the latter). None of the algorithms is fully symbolic. LY and BFH operate on a symbolically-represented transition system and produce an explicit-state minimized transition system. PT is originally expressed for explicit state systems; we converted it to the same hybrid symbolic/explicit style as LY and BFH.

Since model checking is our ultimate goal, our minimization algorithms should terminate early if they discover that the invariant property fails. In other words, we wish to use the bisimulation minimization algorithms as on-the-fly model checkers. In the sections that follow, we describe both the original algorithms and the algorithms we derived from them to support early termination. Our algorithms have the same spirit as the originals, and generally add no more than an extra flag to each block to aid in detecting failed properties. A system that has been minimized around an invariance contains one reachable block if the property holds. In essence, we use this criterion to support early termination. We prove that our algorithms behave as the originals when the tested property holds, and report failure if the tested property does not hold. Our comparisons to BR are with respect to the new algorithms. The correspondence proofs between the new and original algorithms allow us to extrapolate the results from our comparisons to the original algorithms as well.

Sections 5.1, 5.2, and 5.3 discuss the original and new LY, BFH, and PT algorithms, respectively. We discuss the algorithms in this order because LY bears the closest correspondence to BR, followed by BFH, followed by PT.

```
stack := {⟨B, init⟩}, where B is the initial block                                   (1)
mark ⟨B, init⟩                                                                      (2)
queue := ∅                                                                          (3)
partition contains one block per combination of atomic propositions                 (4)

search:                                                                             (5)
  while stack ≠ ∅ do begin                                                          (6)
    ⟨B, p⟩ := pop(stack)                                                            (7)
    D := post(B)                                                                    (8)
    foreach block ⟨C, q⟩ containing a state in post(p) do begin                     (9)
      if B contains a state with no successor in C then enqueue ⟨B, p⟩              (10)
      if ⟨C, q⟩ is unmarked then select q from C ∩ post(p) and push ⟨C, q⟩ onto the stack   (11)
      add edge ⟨B, p⟩ → ⟨C, q⟩                                                      (12)
      D := D − C                                                                    (13)
    end                                                                             (14)
    if D ≠ ∅ then enqueue ⟨B, p⟩                                                    (15)
  end                                                                               (16)

split:                                                                              (17)
  while queue ≠ ∅ do begin                                                          (18)
    ⟨B, p⟩ := delete (queue)                                                        (19)
    B′ := {q ∈ B : blocks(post(q)) = blocks(post(p))} (see Figure 4 for the code that does this)   (20)
    B″ := B − B′                                                                    (21)
    B := B′                                                                         (22)
    add B″ to the partition                                                         (23)
    foreach edge ⟨C, q⟩ → ⟨B, p⟩ in the minimized system do begin                   (24)
      if ∅ ≠ C ∩ pre(B) ≠ C or ∅ ≠ C ∩ pre(B″) ≠ C then enqueue ⟨C, q⟩              (25)
      if post(q) ∩ B = ∅ then delete edge ⟨C, q⟩ → ⟨B, p⟩                           (26)
      if post(q) ∩ B″ ≠ ∅ then                                                      (27)
        if the block for B″ is not marked then                                      (28)
          select p″_b in post(q) ∩ B″, mark ⟨B″, p″_b⟩, and push ⟨B″, p″_b⟩ onto stack   (29)
        add edge ⟨C, q⟩ → ⟨B″, p″_b⟩                                                (30)
      endif                                                                         (31)
    end                                                                             (32)
    if stack ≠ ∅ then goto search                                                   (33)
  end
```

Figure 3: The original Lee-Yannakakis combined bisimulation and reachability algorithm [30]. Tuples $\langle C, q\rangle$ represent blocks, where C is the set of states and q is the representative of the block. The representative is undefined until a block is determined to be reachable. The enqueue operation does not insert a block that is already in the queue.

```
B′ := B                                                                             (34)
D := post(B)                                                                        (35)
foreach edge ⟨B, q⟩ → ⟨C, q⟩ in the marked graph do begin                           (36)
  B′ := B′ ∩ pre(C)                                                                 (37)
  D := D − C                                                                        (38)
end                                                                                 (39)
B′ := B′ − pre(D)
```

Figure 4: Computing the states that agree with the representative on the next-state transitions in the Lee-Yannakakis algorithm. This expands the definition of B' in line 20 of Figure 3.

(1) queue := $\emptyset$;
(2) partition contains the good block and the bad block
(3) **let** $\langle B, init\rangle$ be the initial block
(4) **if** $\langle B, init\rangle$ is the bad block **then** signal safety violation and terminate

(5) search: $D := post(B)$
(6) **foreach** block $\langle C, q\rangle$ containing a state in $post(init)$ **do begin**
(7) **if** $\langle B, init\rangle \neq \langle C, q\rangle$ **then** signal safety violation and terminate
(8) **if** $B \cap pre(C) \neq B$ **then** enqueue $\langle B, init\rangle$
(9) add edge $\langle B, init\rangle \rightarrow \langle C, q\rangle$
(10) $D := D - C$
(11) **end**
(12) **if** $D \neq \emptyset$ **then** enqueue $\langle B, init\rangle$

(13) split: **while** queue $\neq \emptyset$ **do**
(14) $\langle B, init\rangle$:= delete (queue)
(15) $B' := (B \cap pre(B)) - pre(post(B) - B)$
(16) $B'' := B - B'$
(17) $B := B'$
(18) add B'' to the partition
(19) **if** $\emptyset \neq B \cap pre(B) \neq B$ or $\emptyset \neq B \cap pre(B'') \neq B$ **then** enqueue $\langle B, init\rangle$
(20) **if** $post(init) \cap B = \emptyset$ **then** delete edge $\langle B, init\rangle \rightarrow \langle B, init\rangle$
(21) **if** $post(init) \cap B'' \neq \emptyset$ **then** signal safety violation and terminate
(22) **endwhile**

Figure 5: The new Lee-Yannakakis algorithm, with early termination for model checking invariant properties.

5.1 Lee-Yannakakis (LY)

The LY algorithm (Figures 3 and 4) stabilizes only reachable blocks. Accordingly, it must compute which blocks are reachable. To assist in this effort, each reachable block has a representative that is guaranteed to be a reachable state. The initial block has the initial state as its representative. At the start of the algorithm, only the initial block is known to be reachable. If there exists a transition from the representative of a reachable block A to a state in an unreachable block B, block B becomes reachable. When the algorithm determines that B is reachable, it chooses a representative for B from among B's known reachable states. The algorithm contains two loops, as shown in Figure 3. The search loop (line 5) uses a depth-first, stack-based search to look for new reachable blocks. The split loop (line 17) stabilizes unstable reachable blocks. Each loop can identify new blocks that the other loop must process. The algorithm terminates when all reachable blocks are stable and no new blocks are reachable.

LY gives searching precedence over splitting; adapting LY to support early termination in invariance checking is therefore straightforward. With the exception of the initial block, all blocks that the algorithm generates have paths to the bad block; thus the on-the-fly algorithm should terminate early if a second block becomes reachable. Therefore, when a new block is pushed onto the search stack (lines 11 and 29), the algorithm should raise a violation and terminate. Given that only one block can be reachable if the property holds, the minimized transition system has at most the edge from the initial block to itself.[2] Taking this observation into account, the pseudocode for the new LY algorithm appears in Figure 5. The split loop repeatedly stabilizes the initial block until either a second block becomes reachable or the initial block is stable. LY henceforth refers to the new algorithm.

This process of repeated stabilization in LY seems quite similar to the repeated generation of frontier sets in BR. To establish a formal correlation between the two algorithms, we want to prove that the set of states removed from the initial block at each iteration of the LY split loop is the same as the frontier set from the corresponding iteration of BR. Inside the split loop, LY creates a new block consisting of those states that reach outside of the current initial block. The remaining states form the new initial block by construction. Figure 6 depicts this process, where set BS_i is the contents of the initial block at the end of iteration i (BS stands for "block states"). We can derive the equation for BS_i

[2] Given this, we could eliminate this edge, and thus the minimized transition relation as well. However, we store the edge in order to maintain the correspondence between this algorithm and the original LY algorithm.

$$BS_0 = Good \quad (5)$$
$$BS_{i+1} = (BS_i \cap pre(BS_i)) - pre(post(BS_i) - BS_i) \quad (6)$$

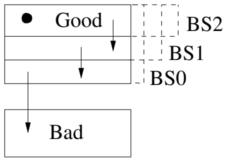


Figure 6: The definition of the reachable block BS_i and how LY carves up the state space.

shown in Figure 6 from the pseudocode in Figure 5. We obtain this equation as follows: if the algorithm reaches the split loop, then the initial state must have been a good state (line 4). This gives the definition of BS_0. If the algorithm is in the split loop and reaches the start of the split loop again, the new definition of B in $\langle B, init\rangle$ comes from line 15, which gives the equation for BS_{i+1} in terms of BS_i.

We wish to prove that the difference between sets BS_{i-1} and BS_i in LY is the set of frontier states F_i computed in BR. In addition, we want to prove that BS_i and S_i are complements of one another. In order to simplify the proof, we first simplify the equation from Figure 6 for computing BS_i to the following:

$$BS_0 = Good \quad (7)$$
$$BS_{i+1} = (BS_i \cap pre(BS_i)) - pre(\overline{BS_i}) \quad (8)$$

The following lemma justifies that this simplification is valid.

Lemma 3 *Let A' be a set, A be a subset of A', and assume there is a transition relation defined from A' to A'. Then $(A \cap pre(A)) - pre(post(A) - A) \equiv (A \cap pre(A)) - pre(\overline{A})$.*

Proof The lemma statement is equivalent to

$$(A \cap pre(A)) - pre(post(A) - A) \equiv (A \cap pre(A)) - (A \cap pre(\overline{A})).$$

Clearly, the lemma holds if

$$pre(post(A) - A) \equiv (A \cap pre(\overline{A})),$$

which asks whether the pre operation distributes over the expression $post(A) - A$.

This distribution is not equivalence preserving in the general case. It can fail if there exists some element x that is in $A \cap pre(\overline{A})$, but that is not in $pre(post(A) \cap \overline{A})$. More specifically, if it fails, then the universe must contain elements x, y_1, and y_2 such that

1. (x, y_1) and (x, y_2) are both in the transition relation,
2. y_1 is in $post(A)$ but not in $\overline{A}$,
3. y_2 is in $\overline{A}$, but not in $post(A)$, and
4. x reaches no element in $post(A) \cap \overline{A}$.

Since y_2 is not in $post(A)$, x cannot be in A (since (x, y_2) is in the transition relation). Therefore, this result can only fail on elements x that are not in A. However, each side of the expression in the lemma statement concerns only those values that are in A. We therefore do not care what happens to values of x that are not in A. The distribution is equivalence preserving for all values of x in A, so the lemma holds. □

We are now ready to prove the correspondence between the sets BS_i that LY computes and the sets S_i and F_i computed during BR.

Lemma 4 *For all iterations i of LY's split loop, $S_i = \overline{BS_i}$ and $F_{i+1} = BS_i - BS_{i+1}$.*

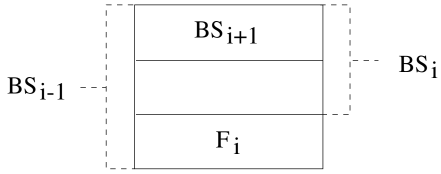


Figure 7: Diagram used in proof of Lemma 4, showing how the LY algorithm carves the set BS_{i-1} to create sets BS_i and BS_{i+1}.

Proof The proof is by induction on i. The base cases cover each expression when i is zero. By definition, S_0 is the set of bad states and BS_0 is the set of good states. By definition, the sets of good and bad states are disjoint and partition the state space. $S_0 = \overline{BS_0}$ therefore holds by definition.

For the other base case, we must prove that $F_1 = BS_0 - BS_1$. The argument consists of the following chains of expressions:

$$\begin{aligned} F_1 &= S_1 - S_0 \\ &= (S_0 \cup pre(F_0)) - S_0 \\ &= (bad \cup pre(bad)) - bad \\ &= pre(bad) - bad \\ &= pre(bad) \cap good \end{aligned}$$

$$\begin{aligned} BS_0 - BS_1 &= good \cap \overline{BS_1} \\ &= good \cap \overline{(good \cap pre(good) - pre(\overline{good}))} \\ &= good \cap \overline{(good \cap pre(good) - pre(bad))} \\ &= good \cap (\overline{good} \cup \overline{pre(good)} \cup pre(bad)) \\ &= good \cap (\overline{pre(good)} \cup pre(bad)) \\ &= good \cap pre(bad) \qquad (*) \\ &= F_1 \end{aligned}$$

The step marked (*) in the derivation of $BS_0 - BS_1$ follows since the transition relation is total. Each state not reaching a state in *good* must reach a state in *bad* since *good* and *bad* partition the state space.

For the inductive case, assume $S_{i-1} = \overline{BS_{i-1}}$ and $F_i = BS_{i-1} - BS_i$. We want to prove that $S_i = \overline{BS_i}$ and $F_{i+1} = BS_i - BS_{i+1}$. We consider each expression in turn. First, consider $S_i = \overline{BS_i}$. There are two cases.

1. If x is in S_i, then by definition, x is in $S_{i-1} \cup pre(F_{i-1})$.
   - If x is in S_{i-1}, then x is not in BS_{i-1} by the inductive hypothesis. However, by definition, BS_i is a subset of BS_{i-1}, so x cannot be in BS_i.
   - Now assume x is in $pre(F_{i-1})$, but not in S_{i-1}. Then x is in $pre(F_{i-1}) - S_{i-1}$, so x is in F_i by definition. By the inductive hypothesis, x is in $BS_{i-1} - BS_i$, so it cannot be in BS_i.
2. If x is not in BS_i, there are two cases to consider.
   - If x is not in BS_{i-1}, then by the inductive hypothesis, x is in S_{i-1}, so x is in S_i by definition.
   - If x is in BS_{i-1}, then it is also in $BS_{i-1} - BS_i$, and hence in F_i by the inductive hypothesis. It follows from the definitions in Figure 2 that $F_i = S_i - S_{i-1}$, so x must be in $S_i - S_{i-1}$. Therefore, x is in S_i.

Next, consider $F_{i+1} = BS_i - BS_{i+1}$. Again, there are two cases to consider.

1. Assume x is in $BS_i - BS_{i+1}$. We need to show that x is in F_{i+1}, which is equivalent to showing that x is in $pre(F_i)$, but not in S_i. Since x is in BS_i, it cannot be in S_i, by the argument given earlier in this proof. Thus, we only need to prove that x is in $pre(F_i)$.

   Since x is in BS_i, then by equation 8, x is in $(BS_{i-1} \cap pre(BS_{i-1})) - pre(\overline{BS_{i-1}})$. From the diagram in Figure 7, since x is in $pre(BS_{i-1})$, x must reach one of the three cells in the figure. If x is not in $pre(BS_i)$, then x must be in $pre(F_i)$ according to the diagram. If x is in $pre(BS_i)$, then x is also in $BS_i \cap pre(BS_i)$. By assumption, x is not in BS_{i+1}. Therefore, x must be in $pre(\overline{BS_i})$, else it would be in BS_{i+1} by equation 8. By the diagram in Figure 7, x therefore must be in $pre(F_i)$.

2. Assume x is in F_{i+1}. Then x is in $pre(F_i)$, but not in S_i. We must show that x is in BS_i but not in BS_{i+1}. Since x is not in S_i, it must be in BS_i by the earlier argument in this proof. Therefore, x must lie in one of the upper two cells in Figure 7. If x were in BS_{i+1} (the uppermost cell), then by definition (equation 8) it could not be in $pre(\overline{BS_i})$. It is clear from the figure that F_i is a subset of $\overline{BS_i}$., so x cannot be in $pre(F_i)$. Since this contradicts the assumptions for this case, x must lie in the middle cell, which is precisely $BS_i - BS_{i+1}$.

□

Given this correlation, we expect LY and BR to perform the same number of iterations before detecting whether an invariant property holds. Given the nature of when the termination checks are performed, LY actually finishes in one iteration fewer than BR (unless neither does any iterations). The following collection of lemmas proves this (the missing proofs appear in the appendix).

Lemma 5 *For $i > 1$, LY starts the $i + 1^{\text{st}}$ iteration of its split loop (Figure 5, line 13) iff $pre(F_i) \cap \overline{S_i} \neq \emptyset$ and $post(init) \cap F_i = \emptyset$.*

Lemma 6 *If LY is in iteration i of the split loop and starts iteration $i+1$ of the split loop, then BR also enters iteration $i + 1$.*

Lemma 7 *If LY terminates in the i^{th} iteration of the split loop, where $i > 0$, then BR terminates in iteration $i + 1$.*

Proof Assume LY starts iteration i, but terminates before reaching iteration $i + 1$. By Lemma 6, BR also enters iteration i. Since LY didn't reach iteration $i + 1$, $pre(F_i) \cap \overline{S_i} = \emptyset$ or $post(init) \cap F_i \neq \emptyset$ by Lemma 5. We consider each case in turn. If $pre(F_i) \cap \overline{S_i} = \emptyset$, then F_{i+1} is empty by definition, so BR terminates in iteration $i + 1$. If $post(init) \cap F_i \neq \emptyset$, then $init$ is in $pre(F_i)$. Furthermore, $init$ cannot be in S_i, or BR would have terminated before entering iteration i. By definition, then, $init$ is in F_{i+1}, so BR terminates in iteration $i + 1$. □

Since BR and LY compute the same sets and require almost the same number of iterations, we would expect them to do roughly the same amount of work. This is not the case, however, because the two algorithms compute the frontier sets using different approaches. BR uses the current frontier states F_i, from which only one image computation is needed to compute F_{i+1}. In the spirit of computing with only reachable blocks, LY must compute F_{i+1} using only those states in BS_i. According to the equation for BS_{i+1} in Figure 6, this requires three image computations. LY therefore does more work to compute the frontier sets than does BR.

The computation lower bound for LY is evident from the pseudocode in Figure 5. Line 15 does $3M + I + 2D$. Line 16 adds D. Line 19 performs one or two $M + I + 2E$ computations[3]. Lines 20 and 21 contribute $M + 2I + 2E$. Assuming line 19 performs only one of its two comparisons, LY does a minimum of $5M + 4I + 3D + 4E$ computations per iteration. Since LY does only one fewer iteration than BR by Lemma 7, we conclude that LY does a non-trivial factor more computations than does BR. Although one cannot prove that LY requires more time or memory than BR, given the differences in the intermediate BDDs used by each algorithm, the sheer increase in image computations that LY performs suggests that this will be the case in the context of BDDs. Our experimental results in Section 6 confirm this.

We conclude this section with proofs that the new LY algorithm is a valid on-the-fly model checker, and that it corresponds to the original algorithm for properties that hold. Let LY_{ET} be the early termination version of LY given in Figure 5. Let LY be the original LY pseudocode given in Figures 3 and 4.

Lemma 8 *LY_{ET} terminates with a safety violation iff the given invariant property fails.*

Proof We consider each direction in turn. Assume LY_{ET} terminates with a safety violation. Then it must have terminated on one of lines 4, 7, or 21 in Figure 5. If it terminated on line 4, then the initial state is bad and the property must fail. If it terminated on line 7, then $post(init)$ must have included a state in the bad block since the search code is only executed once (on the good block) and there are only two blocks in the partition (the good block and the bad block) at that point. The property must fail in this case since the initial state can reach a bad state. If the algorithm terminated on line 21, then $post(init)$ includes some state in set B'', as computed on line 16. B'' corresponds to $BS_i - BS_{i+1}$, which is equivalent to the frontier set F_{i+1} from BR by Lemma 4. By construction, the states in each

[3]The computations of $pre(B)$ on lines 15 and 19 are different because B changed on line 17.

frontier set reach the bad state. Since $init$ can reach some state in this frontier, $init$ also has a path to a bad state, so the property must fail.

Now, assume the invariant property fails. Then there must be a path from the initial state $init$ to some bad state. We consider three cases depending on the length of that path:

1. If the path has length 0, then the initial state is a bad state. In this case, the algorithm terminates with a safety violation on line 4.

2. If the path has length 1, then the initial state can directly reach a bad state. In this case, the bad block will be processed in the **foreach** loop at line 6 during the search from the good block starting at line 5. Since the good and bad blocks are distinct, the algorithm terminates on line 7 with a safety violation.

3. Assume the path has length i, where $i > 1$. Then by construction, $init$ must be in the frontier set F_i from BR, and $init$ must reach a state in F_{i-1}. By Lemma 4, F_{i-1} corresponds to $BS_i - BS_{i+1}$. By definition of the sets BS_i then, F_{i-1} corresponds to set B'' in the i^{th} iteration of the LY_{ET} split loop. Therefore, $post(init)$ must contain a state in B'', so the algorithm terminates on line 21 with a safety violation.

□

Lemma 9 *Assume LY_{ET} and LY start with the same initial partition. If LY_{ET} reaches the split loop for the i^{th} time, it has the same partition, queue contents, and minimized transition relation as LY does upon reaching the split loop for the i^{th} time. Furthermore, once LY reaches line 8 for the first time, the stack becomes non-empty only if the given invariant property fails to hold.*

Proof The proof is by induction on i. Since both algorithms start with the same initial partition, the lemma holds at iteration 0. The block $\langle B, init\rangle$ identified in each algorithm is the same by definition. Consider LY (Figure 3). The block popped from the stack in the search loop is $\langle B, init\rangle$. By manual inspection, the code in the search loops of the two algorithms is identical minus line 11 which puts $\langle C, q\rangle$ into the stack. In this first iteration of the search loop, there are two possible values for $\langle C, q\rangle$: the good block and the bad block. If it is the good block, then $\langle C, q\rangle$ is identical to $\langle B, init\rangle$. Since $\langle B, init\rangle$ has been marked (line 2), the **if** statement at line 11 is never taken, so the two algorithms process $\langle C, q\rangle$ in the same way. If $\langle C, q\rangle$ is the bad block, it is pushed onto the stack. However, in this case there must be a transition from $init$ to a state in bad, so the property must fail to hold.

Therefore, if LY_{ET} is still running after the first iteration of the search loop, both algorithms agree that $\langle B, init\rangle$ is in the queue and that the stack is empty. Neither the partition nor the transition relation have been changed. The two algorithms therefore agree on the partition, queue contents, and minimized transition relation at the start of the first iteration through the split loop.

Assume LY_{ET} and LY have the same partition and queue contents at the start of iteration i of their respective split loops. We must show they have the same partition, queue contents, and minimized transition relation after iteration $i+1$, unless the invariant property fails. According to LY_{ET}, the top of the queue is the current $\langle B, init\rangle$. LY_{ET} has created at most one edge in the minimized transition system, namely from $\langle B, init\rangle$ to itself. The first line at which the two algorithms disagree is the one that computes B' (line 15 in LY_{ET}, line 20 in LY). In LY, B' is computed using the code given in Figure 4. Since the two algorithms agree on the minimized transition relations, LY has at most an edge from $\langle B, init\rangle$ to itself. We can therefore flatten the **foreach** loop at line 36 into lines 37 and 38, substituting B for C. Inlining the sequence of resulting definitions for Figure 4, we get the equation for B' given in line 15 of LY_{ET}.

By a similar argument, LY will do at most one iteration of the **foreach** loop at line 24. Since $\langle C, q\rangle$ must be $\langle B, init\rangle$, we can substitute B for C and $init$ for q in the lines in the **foreach** loop (lines 25–31). The resulting code matches that in LY_{ET} until the **if** statement at line 28. The condition at line 28 of LY matches that on line 21 of LY_{ET}. If this condition holds, the property fails by Lemma 8. If this condition does not hold, then each algorithm reaches the end of the split loop without making any additional changes. Since the two algorithms executed the same steps unless the invariant property failed, they must have made the same changes to the queue contents, partition, and transition relation. The inductive case therefore holds. □

5.2 Bouajjani-Fernandez-Halbwachs (BFH)

Like LY, BFH selects reachable blocks to stabilize. The two algorithms differ mainly in how they stabilize reachable blocks. LY stabilizes a reachable block with respect to only reachable blocks, which is sufficient for verification. BFH,

(1) $R = \{[init]_\rho\}; S = \emptyset$
(2) **while** $R \neq S$ **do**
(3)   choose X in $R - S$
(4)   **let** $N = \text{split}(X, \rho)$;
(5)   **if** $N = \{X\}$ **then**
(6) $S := S \cup \{X\}$;
(7) $R := R \cup \{post_\rho(q) | q \in X\}$;
(8)   **else**
(9) $R := R - \{X\}$;
(10)   **if** $\exists Y \in N$ such that $init \in Y$ **then**
(11) $R := R \cup \{Y\}$;
(12) $S := S - \{Y \in S | X \in post_\rho(Y)\}$;
(13) $\rho := (\rho - \{X\}) \cup N$;
(14)   **endif**
(15) **endwhile**

(16) split(X, ρ) :
(17) $N = \{X\}$;
(18)   **foreach** $Y \in \rho$ **do**
(19) $M := \emptyset$;
(20)   **foreach** W in N **do**
(21)   **let** $W_1 = W \cap pre(Y)$;
(22)   **if** $W_1 = W$ or $W_1 = \emptyset$ **then**
(23) $M := M \cup \{W\}$
(24)   **else** $M := M \cup \{W_1, W - W_1\}$;
(25)   **endfor**
(26) $N := M$;
(27)   **endfor**

Figure 8: The original BFH algorithm [5]. Lines 1–15 define the main algorithm; lines 16–27 define the split routine used by the main algorithm. For states X, $post_\rho(X)$ denotes blocks in the partition ρ that contain an element of $post(X)$. $[init]_\rho$ denotes the initial block. R is the set of all blocks known to be reachable. S is the set of all blocks that are stable with respect to the current partition.

$$G_0 = Good \tag{9}$$

$$G_{i+1} = G_i \cap pre(G_i) - (pre(H_1) \cup \ldots \cup pre(H_{|\rho|-1})) \tag{10}$$

(where $H_1 \ldots H_{|\rho|-1}$ are all the blocks in the partition ρ other than G_i)

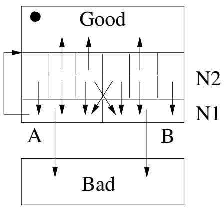


Figure 9: The equations for the unmarked block G_i in BFH and how the algorithm carves up the state space.

on the other hand, stabilizes reachable blocks with respect to all blocks, reachable or unreachable. This results in a somewhat simpler algorithm, at the expense of some unnecessary work.

The BFH algorithm (Figure 8) maintains two lists, one of reachable blocks (R) and one of stable blocks (S). Initially, only the initial block is reachable and no blocks are stable. At each iteration, the algorithm selects a reachable but unstable block and stabilizes it with respect to every block in the partition (lines 3 and 4). If no new blocks are created, the original block was stable. It therefore goes into the stable list and blocks that are reachable from the original block become reachable (lines 5-7). If new blocks are created, the algorithm adds them to the partition, updates the initial block (if necessary), and removes from the stable blocks list those blocks that became unstable as a result of the split (lines 9-13). The algorithm terminates when all reachable blocks are stable.

Figure 9 shows how the BFH algorithm carves up the state space over the first two iterations when checking an invariant property. In the diagram, N_1 labels the slice of states that is removed from the initial block on the first iteration. BFH divides N_1 into two blocks A (left) and B (right). A contains those states that reach both the bad block and the good block. B contains those states that reach only the bad block. N_1 ($A \cup B$) contains the same states as the frontier set F_1 from BR (the same set of states removed during the first iteration of LY). In the second iteration, BFH removes the states in the slice labeled N_2 from the initial block. These are the same states as are in F_2. However, N_2 consists of six disjoint blocks as marked in the diagram, as opposed to the single frontier set of BR or the single new block that LY produces. There is one new block for each combination of blocks to which a state may have next state transitions, as shown by the arrows leaving the blocks in N_2.

Adding early termination to BFH requires a little more work than for LY. As in LY, BFH could terminate when a

```
I = [init]_ρ;
Mark the bad block
while I is not marked do
    N := split(I, ρ);
    if N = {I} then
        if post_ρ(I) − {I} ≠ ∅ then
            Signal safety violation and terminate
        else break ;
        endif
    else
        ρ := (ρ − {X}) ∪ N;
        I := [init]_ρ;
    endif
endwhile
if I is marked then signal safety violation

split(X, ρ) :
  N = {X};
  foreach Y ∈ ρ do
    M := ∅;
    foreach W in N do
      let W_1 = W ∩ pre(Y);
      if W_1 = W or W_1 = ∅ then
        if Y is marked and W = W_1 then mark W
        M := M ∪ {W}
      else
        if Y or W is marked then mark W_1
        if W is marked then mark W − W_1
        M := M ∪ {W_1, W − W_1};
      endif
    endfor
    N := M;
  endfor
  return N
```

Figure 10: The new BFH algorithm supporting early termination for verifying invariant properties. The only new variable over the original algorithm is I, which holds the initial block.

second block becomes reachable. This would correctly determine violations of invariant properties, but not necessarily as soon as they occur. BFH augments the set of reachable blocks only when the block chosen to be split is stable. However, the algorithm may traverse a path from the bad block to the initial state before the initial block becomes stable (we have encountered this case experimentally). Therefore, the algorithm takes extra iterations to terminate in the face of property failure. LY does not take these extra iterations because it tests for reachability within each split iteration, regardless of whether the processed block is stable. We can make BFH terminate as soon as possible by recording whether there is a path from each block to a bad state. We augment each block with a marker flag which is on when every state in the block has a path to a bad state. If the initial block becomes marked, the algorithm can terminate, regardless of whether the initial block is stable. BFH henceforth refers to the new algorithm, which appears in Figure 10.

As for LY, we would like to characterize the subset of the good states that have not yet been found to reach other blocks at the start of each iteration through the split loop. Let G_i denote this set. Figure 9 provides equations for computing G_i based on the BFH pseudocode in Figure 10. The following lemma shows the correspondence between each G_i and the sets BS_i defined for LY.

Lemma 10 *Let BS_i and G_i be defined as in Figures 6 and 9, respectively. For all i up to the number of iterations of LY, $G_i = BS_i$.*

Proof The proof is by induction on i. By definition, both BS_0 and G_0 are the good states, so the base case holds. Assume $G_i = BS_i$. We want to prove that $G_{i+1} = BS_{i+1}$. Consider their definitions.

$$BS_{i+1} = (BS_i \cap pre(BS_i)) - pre(\overline{BS_i})$$

$$G_{i+1} = G_i \cap pre(G_i) - (pre(H_1) \cup \ldots \cup pre(H_{|\rho|-1}))$$

Since $G_i = BS_i$, this is true if $pre(\overline{BS_i}) = (pre(H_1) \cup \ldots \cup pre(H_{|\rho|-1}))$, which is equivalent to asking whether $pre(\overline{BS_i}) = pre(H_1 \cup \ldots \cup H_{|\rho|-1})$. Thus, it suffices to show that $\overline{BS_i} = H_1 \cup \ldots \cup H_{|\rho|-1}$, or equivalently, that $\overline{G_i} = H_1 \cup \ldots \cup H_{|\rho|-1}$. But by definition, $H_1 \cup \ldots \cup H_{|\rho|-1}$ is the union of all blocks in the partition other than G_i. Since the blocks in a partition are disjoint and cover the state space by definition, each state not in G_i must be in $H_1 \cup \ldots \cup H_{|\rho|-1}$ and vice-versa. The lemma therefore holds. □

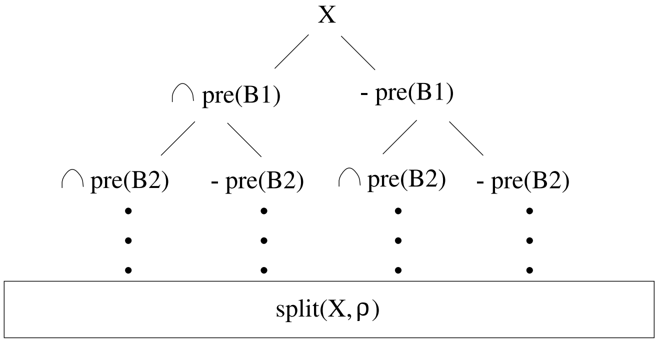


Figure 11: The operation of the BFH stabilization routine. Each path through the tree specifies an expression that yields a subset of the set of states X. The expression is formed by concatenating the expressions found at each node on the path. For example, taking the leftmost branch would yield expression $X \cap pre(B_1) \cap \ldots \cap pre(B_n)$, where n is the number of blocks. BFH computes the sets in this tree using breadth-first search, so only one pre-image set is required at each step through the split algorithm.

This correspondence between G_i and BS_i also suggests a correlation in the number of iterations that these algorithms take before terminating. The next group of lemmas proves that BFH generally terminates after the same number of iterations as does BR.

Lemma 11 *If a block H is marked in BFH, then all states in H have a path to a bad state.*

Lemma 12 *G_i is the only unmarked block in iteration i of the BFH* **while** *loop.*

Lemma 13 *If BR terminates in iteration 0, BFH terminates after at most one call to its split routine. If BFH terminates without executing split, then BFH terminates in iteration 0. For all $i > 0$, BR terminates in iteration i iff BFH terminates after executing its split routine i times.*

Proof BR terminates when $F_i = \emptyset$ or $init \in F_i$. BFH terminates if I (the initial block) is marked (line 15) or I is stable (lines 5-8). Assume i is zero. By definition, F_0 is the set of bad states. If $init$ is in F_0, then by definition, $init$ is in the bad block, so BFH will terminate when it first reaches the **while** loop at line 3. If BR terminates because F_i is empty, then there is no bad block. Thus, since the transition relations are total, BFH will determine that the initial block is stable after one iteration, at which point BFH will terminate. If BFH terminates before calling split, then the initial block was marked the first time BFH reached the **while** loop at line 3. By definition, $init$ must have been in the bad block, so $init$ is in F_0 by definition.

Assume neither BR nor BFH terminated in the i^{th} iteration of the split loop. We must prove that BR terminates in iteration $i+1$ iff BFH terminates in iteration $i+1$. Combining Lemmas 4 and 10, $F_i = G_{i-1} - G_i$. We use this result several times in this proof.

For the forwards direction, we must show that the termination conditions of BR imply the termination conditions for BFH. If $F_i = \emptyset$, then $G_{i-1} - G_i = \emptyset$. In this case, I must be stable. Therefore, the test at line 5 succeeds. Since both branches of the **if** statement at lines 6-8 result in termination, this case holds. BR may also terminate if $init \in F_i$. If $init \in F_i$, then $init$ is in $G_{i-1} - G_i$, so G_{i-1} was unstable (since G_{i-1} does not equal G_i). BFH therefore reaches the **else** clause at line 10. By Lemma 12, G_i is the only unmarked block after iteration i. Since $init$ is not in G_i, $init$ must be in some marked block. Therefore, the next test at line 3 would fail, so BFH would not enter the **while** loop on the $i + 1^{\text{st}}$ iteration.

For the backwards direction, we must show that the termination conditions of BFH imply the termination conditions for BR. BFH terminates under two conditions: if the initial block I is marked or if the initial block is stable. If I is marked, then by Lemma 11, all states in I have a path to a bad state. Since I contains $init$ (by definition), there must be a path from $init$ to a bad state. Had I be marked in the previous iteration, BFH would have terminated. Hence $init$ must be in $G_{i-1} - G_i$ (since G_i is the unmarked block at each iteration by Lemma 12). Therefore, $init$ is in F_i, so BR would also terminate. In the other case, where I is stable after BFH executes the split loop, $G_{i-1} = G_i$, so F_i is empty. BR also terminates in this case. □

We can determine the lower bound for BFH from the pseudocode in Figure 10. The only symbolic computations in the left column of pseudocode occur at line 6, when locating the blocks reachable from the stable initial block. Since the algorithm terminates after executing this step, this code is only executed once and all split iterations occur before this point. The bound therefore focuses on the split loop computations. Over lines 21 and 22 of the split code, the algorithm does $M + I + 2E$ symbolic computations. This code is executed at least once for every block in the current partition ρ. For each new block created, the algorithm also does one difference computation at line 28. The lower bound therefore depends on the number of blocks existing at each iteration. At a minimum, each iteration adds one block (else the initial block is stable, and the algorithm will terminate). In the first iteration, there are two blocks (the good block and the bad block). Iteration j therefore contains at least $j + 1$ blocks.

The lower bound is therefore $\sum_{j=1}^{n}(j+1) * (M + I + 2E) + D$, where n is the number of BFH iterations taken. This is equivalent to $(M + I + 2E) * \frac{n^2+3n}{2} + n * D$. Given the correlation between the numbers of iterations of BR and BFH from Lemma 13, this is an order of magnitude more image computations than the $O(n)$ image computations performed by either BR or LY. Furthermore, in this minimal case, the sets on which the images are computed across BR and BFH would be identical. (However, as the number of sets gets larger, the sizes of the sets for which BFH computes images gets smaller, which may well offset the cost of the extra computations). We therefore expect BFH to require more resources than BR, particularly as the number of iterations increases. For similar reasons, we also expect BFH to require more resources than LY on examples requiring many iterations. The experiments reported in Section 6 explore these hypotheses.

We conclude this section with proofs that the new BFH algorithm is a valid on-the-fly model checker, and that it is faithful to the original algorithm. Let BFH_{ET} be the early termination version of BFH given in Figure 10. Let BFH be the original BFH pseudocode given in Figure 8.

Lemma 14 *BFH_{ET} terminates with a safety violation iff the given invariant property fails.*

Proof We consider each direction in turn. Assume BFH_{ET} terminates with a safety violation. Then it does so on one of lines 7 and 15 of Figure 10. If it terminated on line 15, then initial block was marked; by Lemma 11, the initial state must have had a path to a bad state, so the property fails. If the algorithm terminated on line 7, then the condition on line 6, namely $post_\rho(I) - \{I\} \neq \emptyset$, must have held. This condition tests whether the initial block, which must be stable to have passed the test on line 5, reaches states in some other block. If this condition passed, then the initial state must reach some state in another block; that other block must be marked, and hence indicate a path to a bad state, by Lemmas 11 and 12. The property therefore fails.

Now assume the property fails. Then by definition there exists a path from the initial state to a bad state. We consider two cases depending upon the length of that path:

1. If the path has length 0, then the initial state is a bad state by definition. In this case, BFH_{ET} marks the initial block at line 2 and fails to enter the **while** loop at line 3. It goes to line 15 and terminates with a safety violation.

2. Assume the path has length i where $i > 0$. Then by construction, $init$ must be in the frontier set F_i from BR, and $init$ must reach a state in F_{i-1}. Combining Lemmas 4 and 10, $F_i = G_{i-1} - G_i$. Since $init$ is in $G_{i-1} - G_i$, G_{i-1} must have been unstable. BFH_{ET} therefore reaches the **else** clause at line 10. By Lemma 12, G_i is the only unmarked block after iteration i. Since $init$ is not in G_i, $init$ must be in some marked block. Therefore, the next test at line 3 would fail, and the algorithm would terminate with a safety violation at line 15.

□

Lemma 15 *If the invariant property holds, then after each iteration i of its* **while** *loop (line 3), BFH_{ET} has the same partition as BFH does after iteration i of its* **while** *loop (line 3). Furthermore, the initial block is the only block in the set of reachable blocks (R) in BFH and the set S in BFH is empty until the final iteration.*

Proof The proof is by induction on i. In the base case, i is 0. Both algorithms start with the same initial partition consisting of the good and bad blocks. The initial block is the only reachable block, so R in BFH contains I; S in BFH is empty.

Assume the two algorithms have the same partitions at the start of their i^{th} iterations, that R in BFH contains only the initial block, and that S in BFH is empty. Then at line 3, BFH must choose the initial block as block X. Each algorithm calls its split routine on the initial block. The code for the split routines is the same across the two algorithms

The Paige-Tarjan algorithm:

1. (Select a refining block) Remove some block S from C (block S is a compound block of X). Examine the first two blocks in the list of blocks of Q contained in S. Let B be the smaller (break a tie arbitrarily).
2. (update X) Remove B from S and create a new (simple) block S' of X containing B as its only block of Q. If S is still compound, put S back into C.
3. (refine Q with respect to B) For each block D of Q containing some element of $pre(B)$, split D into $D_1 = D \cap pre(B)$ and $D_2 = D - D_1$ (*i.e.*, replace D in Q with D_1 and D_2). If D_2 is empty, eliminate it from Q. If it is non-empty and the block of X containing D_1 and D_2 has been made compound by the split, add it to C.
4. (refine Q with respect to $S - B$) For each block D of Q containing some element of $pre(B) - pre(S - B)$, split D into $D_1 = D \cap pre(B) - pre(S - B)$ and $D_2 = D - D_1$ (*i.e.*, replace D in Q with D_1 and D_2). If D_2 is empty, eliminate it from Q. If it is non-empty and the block of X containing D_1 and D_2 has been made compound by the split, add it to C.

Figure 12: The original Paige-Tarjan algorithm [32], minus details relevant only to their data structures for nodes and edges. Q is the current partition. X is a previous partition. Q refines X. C is the set of compound blocks of X.

with the exception of the lines that mark blocks in BFH_{ET}. These lines do not affect the partitions, so the two split routines return the same partitions in terms of block contents. There are then two cases to consider, depending on whether the initial block was stable.

1. If the initial block was stable, then the **if** tests at line 5 in each algorithm passes. In BFH, the initial block goes into the list S of stable blocks. Since the property holds, no new blocks are added to R at line 7. The sets R and S now contain the same blocks, so BFH terminates without changing the partition again. Since the property holds, BFH_{ET} reaches the **break** statement on line 8 and terminates without changing the partition again. The two algorithms therefore terminate with the same partition.
2. If the initial block was not stable, then BFH removes the initial block from R at line 9 and puts it back into R at line 11. R therefore contains only the initial block at the end of this iteration. BFH doesn't add anything to S, so S remains empty. Both BFH and BFH_{ET} update their partitions ρ in the same way on lines 13 and 11, respectively, so the partitions are also the same at the end of the i^{th} iteration.

□

5.3 Paige-Tarjan (PT)

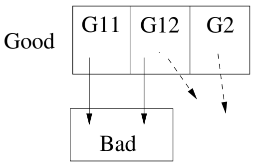
Unlike LY and BFH, PT does not distinguish between reachable and unreachable blocks [32]. Instead, it stabilizes every block with respect to all blocks. PT is interesting to this study because, unlike LY and BFH which choose blocks to stabilize, PT chooses splitters and stabilizes the entire system with respect to each splitter. This bounds how many times each state appears in a splitter, which leads to the low ($n \log n$, where n is the number of states) worst-case running time of the PT algorithm compared to other (explicit-state) minimization algorithms. To aid in choosing splitters, PT maintains two partitions: the current partition Q and a previous partition X which refines Q, as shown in the pseudocode in Figure 12. PT chooses splitters by looking for blocks of X that contain states from multiple blocks of Q; such blocks of X are called *compound*. After selecting a compound block, PT uses the enclosed block with the smallest number of states as the splitter. The algorithm terminates when $Q = X$, which indicates that all blocks are stable.

We modify PT to support early termination by adding bad path markers to blocks, as we did for BFH; the new algorithm appears in Figure 13. On each iteration, if the splitter B is marked, then we propagate markings to all blocks that reach B. Our new PT algorithm contains at most one unmarked block at each iteration. If the splitter is not marked, we can propagate markings to all blocks that do not reach the splitter (because they must reach some other, marked, block). As in BFH, early termination occurs if the initial block becomes marked. Henceforth, PT refers to the new algorithm.

The algorithm in Figure 13 includes an additional modification: it never splits marked blocks. For checking invariant properties, we only care about whether the initial block can reach a marked block. Therefore, we want to stabilize the initial block with respect to other (marked and unmarked) blocks. Stabilizing a block A with respect

initialize X to $\{\mathcal{U}\}$
initialize Q to the initial partition $\{Good, Bad\}$
set the initial block to the block containing the initial state
mark Bad
while $Q \neq X$ and the initial block is not marked **do**
  select some compound block S from X
  let B be the block in S with the smallest number of states
  remove B from S and add block S' containing only B to X
  $E_1 = pre(B)$
  $E_2 = E_1 - pre(S - B)$
  foreach block D of Q that contains an element of E_1 and is unmarked
    $D_1 = D \cap E_1$
    $D_2 = D - D_1$
    Replace D in Q with block D_1; re-direct pointer to D in X to D_1
    if B is marked **then** mark D_1 **endif**
    if D_2 is non-empty **then**
      add D_2 to Q
      if B is not marked **then** mark D_2 **endif**
      add a pointer to D_2 within block of X containing D_1
    endif
    if D_1 contains an element of E_2 **then**
      $D_{11} = D_1 \cap E_2$
      $D_{12} = D_1 - D_{11}$
      Replace D_1 in Q with block D_{11}; re-direct pointer to D_1 in X to D_{11}
      if B is marked or D_1 is marked **then** mark D_{11} **endif**
      if D_{12} is non-empty **then**
        add D_{12} to Q
        add a pointer to D_{12} within block of X containing D_{11}
        mark D_{12}
      endif
    endif
    if D was the initial block **then**
      set the initial block to the block containing the initial state
  endfor
endwhile
if the initial block is marked **then** signal safety violation

Figure 13: The new Paige-Tarjan algorithm, supporting early termination for verifying invariant properties. Unlike the original, it also handles symbolically-represented transition systems and does not split marked blocks.



$$X_0 = \{\mathcal{U}\}$$
$$Q_0 = \{Bad, Good\}$$

$$X_1 = \{Bad, Good\}$$
$$Q_1 = \{G_{11}, G_{12}, G_2\}$$

Figure 14: How the new PT algorithm carves up the state space on the first iteration. Q is the current partition. X is the previous partition used in the selection of splitters. $\mathcal{U}$ denotes the entire state space. This diagram assumes that the bad block is chosen as the first splitter. The dashed arrows point to states outside of the bad block (which would be the good block in practice, but this iteration of the algorithm views the split as entirely relative to the bad block).

	B marked	B unmarked
D_1 marked	line 15	
D_2 marked		line 18
D_{11} marked	line 25	
D_{12} marked	line 29	line 29

Table 2: When (and where) the PT early termination algorithm in Figure 13 marks blocks as having paths to bad states. If a cell contains a line number, the block is marked at the indicated line in Figure 13. If a cell in the table is left blank, the algorithm leaves the corresponding block unmarked.

to block a B stabilizes A with respect to any blocks that later split off from B. Therefore, splitting marked blocks does not help stabilize the initial block. This modification leads to substantial savings of computational resources in practice. Furthermore, it is still in the spirit of the original PT, since we have not modified how splitters are chosen.

Figure 14 shows how our new algorithm carves up the state space on the first iteration. As in BFH, the union of blocks G_{11}, G_{12}, and G_2 is the frontier F_1 from BR. However, on subsequent iterations, the new blocks may not contain the entire next frontier set because the chosen splitter may contain only a subset of the previous frontier. For this reason, the correlation between the sets computed in each iteration of PT and BR is less precise than for LY or BFH. Correspondingly, as the following lemmas show, PT may require more iterations than BR to terminate.

Lemma 16 *At the start of every iteration through the PT* **while** *loop (line 5 in Figure 13), at most one block in Q remains unmarked.*

Lemma 17 *PT marks a newly created block only if every state in the block has a path to a bad state.*

Lemma 18 *At the end of iteration i in PT, the unmarked block U_i is stable with respect to B (the splitter) and $S - B$.*

Lemma 19 *Let U_i be the unmarked block at iteration i of PT and let M be a marked block. If U_i and M are in different blocks of X, then U_i is stable with respect to M.*

Lemma 20 *Let $\{U_i, M_1, \ldots, M_k\}$ be the block of X containing the unmarked block U_i at the start of iteration i of PT. If one of $U_i, M_1, \ldots, M_k$ is chosen as the splitter in iteration i, $U_{i+1} \subseteq pre(U_i \cup M_1 \cup \ldots \cup M_k)$.*

Lemma 21 *Let U_i denote the unmarked block at the start of iteration i of PT. Let $\{U_i, M_1, \ldots, M_k\}$ be the block of X that contains U_i. $M_1 \cup \ldots \cup M_k \subseteq F_0 \cup \ldots \cup F_{i-1}$, where each F_j is a frontier set as defined for BR.*

Lemma 22 *At the end of each iteration i of PT, if there exists an unmarked block U_i then $\overline{S_i} \subseteq U_i$, where S_i is the set of explored states from the definition of BR.*

Lemma 23 *If F_i from BR is non-empty and U_{i-1} is stable in PT, then the invariant property being tested must fail.*

Lemma 24 *PT requires at least as many iterations as BR does to terminate.*

Proof First, assume the invariant property holds. PT only terminates when every block in the system is stable. If BR requires n iterations to terminate and PT terminates in iteration j where $j < n$, then U_{j-1} would be stable while F_j is non-empty. Thus the invariant property would fail, which violates our assumption. Therefore, by Lemma 23, PT cannot terminate in fewer iterations than BR.

Now assume the invariant property fails. Then BR terminates because $init \in F_n$ for some $n \geq 0$. Since the frontier sets are all disjoint, F_n must be the first frontier set containing $init$. By definition the of F_n (Figure 2), $init$ must be in $pre(F_{n-1})$. PT reports the invariant property as failed when the initial block becomes marked; this happens when the initial block reaches a marked block. Since the first frontier set $init$ reaches is F_{n-1}, the unmarked block must reach F_{n-1} before the invariant property can be reported to fail. Since the unmarked block is only ever stabilized with respect to the marked blocks in its same compound block of X (by Lemma 19), the element of F_{n-1} that $init$ reaches must be in one of the marked blocks in the compound block holding the marked block. By Lemma 21, this cannot happen until at least iteration n of PT, so PT needs at least n iterations to mark the initial block. □

We derive the lower bound for PT from the pseudocode in Figure 13. The lower bound contains $2M+D$ operations per iteration from lines 9 and 10 of the pseudocode. The membership test on line 11 contributes $I+E$. If the unmarked block has a transition to the chosen splitter, the algorithm does at least $I+D+E$ from lines 12, 13, and 16, and possibly another $I+D+E$ from lines 22, 23, and 26. The membership test on line 21 contributes another $I+E$. In the minimal case, however, the unmarked block would not have a transition to the splitter. The lower bound for PT is therefore $n * (2M + D + I + E)$, where n is the number of **while** loop iterations that PT takes. By Lemma 24, the number of BR iterations is a lower bound for the number of PT iterations. PT will therefore perform at least twice as many image computations as BR, possibly many more. We therefore expect BR to also outperform PT.

We conclude this section with a proof that our new PT is a valid on-the-fly model checker. We also prove a limited correspondence between our version of PT and the original version. This correspondence is weaker than those for LY or BFH due to our optimization of not splitting unreachable blocks in PT. Let $\mathrm{PT}_{\mathrm{ET}}$ be the early termination version of PT given in Figure 13. Let PT be the original PT pseudocode given in Figure 12.

Lemma 25 *$\mathrm{PT}_{\mathrm{ET}}$ terminates with a safety violation iff the given invariant property fails.*

Proof We consider each direction in turn. Assume $\mathrm{PT}_{\mathrm{ET}}$ terminates with a safety violation. The pseudocode (Figure 13) only reports a safety violation on the last line (line 36), and then only if the initial block is marked. By Lemma 17, a block is marked only if every state in it has a path to a bad state. Thus, if the state containing the initial block becomes marked, the initial block has a path to a bad state and the property fails.

Now assume the property does not hold. Then by definition, there exists a path from the initial state to a bad state. We consider three cases depending upon the minimal length of such a path (where the length is the number of edges in the path):

1. If the minimal length is 0, then the initial state is a bad state. The initial state is therefore in the bad block, which is marked at line 4 of $\mathrm{PT}_{\mathrm{ET}}$. Since the initial block is now marked, the algorithm does not enter the **while** loop at line 5, going instead to line 36. The algorithm raises a safety violation on line 36 and terminates.

2. If the minimal length is 1, then the initial state must have a transition to a state in the bad block. The splitter in the first iteration must be either the good block or the bad block, as they are the only two blocks in the initial partition. If the splitter is the good block, then $init$ reaches something in $S - B$, so $init$ goes into either block D_2 or block D_{12}. If the splitter is the bad block, the $init$ reaches something in B, so $init$ goes into block D_1, D_{11}, or D_{12}. In either case, $init$ ends up in a marked block, as shown in Table 2. The initial block is therefore marked at the end of this iteration, and the algorithm will terminate with a safety violation.

3. Assume the minimal length is greater than one. We must prove that $\mathrm{PT}_{\mathrm{ET}}$ eventually marks the initial block. Let $init'$ refer to the second state on this path (the one that $init$ reaches). Since the path has length at least two, both $init$ and $init'$ must be in the good block in the initial partition (otherwise there would be a shorter path from $init$ to a bad state).

   The algorithm cannot terminate with $init$ and $init'$ in the same block. If it did, then $init$ and $init'$ would agree on all of their transitions to other blocks. This would imply a path from $init$ to a bad state that could bypass

$init'$, and thus be shorter than the current minimal length path. Since $init$ and $init'$ start in the same block of the partition and must eventually end up in different blocks of the partition, there must exist an iteration that splits $init$ and $init'$ into separate blocks.

Let i be the first iteration after $init$ and $init'$ have been separated. Let $\{U_i, M_1, \ldots, M_k\}$ be the compound block containing the unmarked block at iteration i. We can assume that the initial block is still unmarked, else we would have terminated with a safety violation. $init$ is therefore in U_i. $init'$ is in some M_j, since any states split off from the unmarked block during a given iteration go into blocks in the same compound block as the unmarked block (lines 19 and 28). Eventually, the algorithm must choose one of U_i or M_j as the splitter. If U_i is the splitter, then $init'$ is in $S - B$. This places $init$ into either D_2 or D_{12}, both of which become marked by Table 2. If M_j is the splitter, then $init$ reaches B, which makes the initial block one of D_1, D_{11}, or D_{12}. Each of these is marked according to Table 2. Thus the initial block must eventually become marked, so the algorithm must terminate with a safety violation.

□

Lemma 26 *If the property holds, then the final partition arising from PT refines the partition arising from PT*$_{\mathrm{ET}}$.

Proof The proof is by induction on the number of iterations through the **while** loop in PT$_{\mathrm{ET}}$. In the base case, the PT$_{\mathrm{ET}}$ partition contains the good block and the bad block. PT started with this same partition. Since the PT algorithm refines its current partition, the PT partition refines the PT$_{\mathrm{ET}}$ partition at the start of the first iteration.

Assume the PT partition refines the PT$_{\mathrm{ET}}$ partition at the start of a given iteration. If the PT partition does not refine the PT$_{\mathrm{ET}}$ partition at the end of this iteration, then there must exist states s_1 and s_2 that are in the same block in the final PT partition, but that are split into different blocks in this iteration of PT$_{\mathrm{ET}}$. Since PT$_{\mathrm{ET}}$ splits s_1 and s_2, s_1 and s_2 must not agree on their transitions to B (the splitter) and $S - B$. The blocks in $S - B$ are distinct from B in the partition. Since the PT partition refines the PT$_{\mathrm{ET}}$ partition before the split, PT must contain distinct blocks that refine B and the blocks in $S - B$ and for which s_1 and s_2 do not agree on their next-state transitions. This implies that the block containing s_1 and s_2 in PT is not stable with respect to these blocks. This is a contradiction, because PT only terminates when all blocks in the partition are stable with respect to one another. The PT partition must therefore refine the PT$_{\mathrm{ET}}$ partition at the end of this iteration. □

6 Experimental Comparisons

In the previous section, we developed a lower bound on the numbers of various computations that each algorithm (BR, LY, BFH, and PT) performs. We also showed that the number of BR iterations exceeds the number of LY iterations by one (Lemma 7), and that it provides a lower bound for the number of iterations taken in BFH and PT (Lemmas 13 and 24). Combining these results yields the following lower bounds for each algorithm, where n is the number of BR iterations:

BR: $n * (M + U + D + 2E + I)$

LY: $(n-1) * (5M + 4I + 3D + 4E)$

BFH: $(M + I + 2E) * \frac{n^2+3n}{2} + n * D$

PT: $n * (2M + D + I + E)$

These bounds suggest that BR should have the best performance. The bound for LY is so much more expensive than the other algorithms with respect to image computations because LY splits only reachable blocks with respect to reachable blocks. This requires it to calculate frontier sets in a less straightforward fashion than the other algorithms, which will perform computations with unreachable blocks. In the context of symbolic model checking using BDDs, the image computations are typically the most expensive. Therefore, to estimate performance with respect to BDDs, we should compare algorithms with respect to the number of image computations (variable M). Consider LY versus BFH. If n is less than five, BFH's minimum number of image computations is smaller than LY's. The two algorithms agree on the number of image computations if n is five. LY's minimum number of image computations is better if n is greater than five. Therefore, we expect BFH to perform no worse on designs that terminate on small numbers of iterations, but we also expect that performance to degrade as the number of iterations gets larger. It is hard to predict PT's performance because we cannot reasonably predict how many more iterations it will do than the other algorithms.

Table 3 presents the time and memory usage statistics for running these algorithms on a suite of invariant properties, some of which hold and some of which do not. Our experimental framework uses the VIS model checker (version

	State	BR			LY		BFH		PT		
	Vars	Iter	Time	Mem	Time	Mem	Time	Mem	Iter	Time	Mem
gigamax	16	6	1.8	5.59	2.3	5.60	2.1	5.59	6	2.0	5.58
eisenberg*	17	19	1.1	3.80	3.3	3.99	9.3	4.48	270	9.3	4.28
abp	19	11	0.9	3.81	2.3	3.85	2.8	3.82	19	2.0	3.86
bakery*	20	58	1.3	3.70	6.5	3.87	126.9	9.67	212	7.8	4.55
treearb4	23	24	3.5	4.28	16.9	5.18	99.0	6.14	232	118.3	6.06
elev23	32	1	3.9	8.45	4.2	8.54	4.4	8.51	1	4.0	8.43
coherence1	37	5	3.6	6.28	85.5	22.0	33.0	20.0	23	29.5	8.55
coherence2	37	14	9.3	7.81	279.3	31.0	174.8	21.0	166	567.4	18
coherence3*	37	5	6.5	7.96	84.2	20.0	24.4	11.0	9	7.9	7.89
coherence4*	37	5	7.3	8.58	78.2	18.0	34.4	11.0	685	13.8H	68
elev33	45	1	7.0	11.0	444.5	17.0	443.8	17.0	1	7.2	11
elev43	56	1	11.9	15.0	1590.1	42.0	1661.0	39.0	1	12.2	15
tcp*	80	1	3.1	7.83	3.6	8.06	3.0	8.08	1	3.2	7.83

Table 3: Experimental comparison of the various algorithms. A $*$ after an experiment name indicates that the tested invariant property fails. The Iter columns indicates how many iterations an algorithm took before locating an error or reaching a fixpoint. We do not provide these columns for LY and BFH because we can compute their iterations from those for BR by Lemmas 7 and 13 (we have confirmed those lemmas experimentally). The units for the Time and Mem columns are seconds and megabytes, respectively. An H after the time indicates time in hours, rather than seconds.

1.2) [35] as a front-end to obtain the BDDs for the transition relation, initial state, and initial partition from a given Verilog design and a CTL invariant property. We then feed these BDDs into our own code for each of the tested algorithms. This code uses VIS's routines for performing image computations (using partitioned transition relations), and for interfacing to the CUDD BDD package. All of the designs and their tested properties are from the VIS distribution. In order to have precise control over the BR experiments, we used our own implementation of BR, rather than the built-in VIS model checker. All runs were performed on an UltraSparc 140 with 128 megabytes of memory running Solaris 5.5.1.

Table 3 shows that BR has better time and memory usage than the three adapted minimization algorithms in all cases with the exception of elev23, in which PT achieved a negligible savings in memory over BR. This supports our hypothesis that bisimulation minimization plus model checking requires more resources than model checking alone. These figures also let us compare the minimization algorithms. As predicted, BFH does perform worse than LY on the designs that require the largest numbers of iterations, namely eisenberg, bakery, and treearb4. The difference between BFH and LY is most pronounced for bakery, which required the most iterations. BFH generally requires less memory and time than LY on the smaller examples. The one example that falls outside of our predictions is coherence2. With 14 iterations, we predicted LY would perform better, which it does not. The discrepancy is particularly startling when we look at the number of image computations that each algorithm performs, as shown in Table 4. For coherence2, LY does only 78 image computations, as opposed to 3058 by BFH. We conclude that BFH computed images of much smaller sets (since the two algorithms do almost the same number of iterations), and hence was able to avoid blowup in the intermediate BDDs used in the image computations. A similar explanation might also account for BFH's superior performance on examples coherence3 and coherence4. According to Alur *et al.*, who compared the performance of the original LY and BFH algorithms on a small number of timed automata, BFH performed better on their examples [1]. It would be interesting to know how many iterations and image computations their examples required, and whether these figures might explain their results.

The PT algorithm does surprisingly well in comparison to BFH and LY. With the exception of example coherence4, PT has comparable or better memory performance. Its time performance is comparable or better on all examples but treearb4, coherence2, and coherence4. Thus, choosing unreachable blocks as splitters does not appear to hurt PT's performance. However, our version of PT splits only reachable blocks. To measure the effect of this optimization, Table 5 shows the iteration, time, and memory requirements for PT without this optimization. It is clear from this table that the restriction to stabilizing reachable blocks makes a substantial difference to PT's success. That PT performs so well compared to BFH and LY suggests that minimization algorithms tailored to verification settings should pay

	BR Iterations	LY		BFH		PT	
		lower bound	actual	lower bound	actual	lower bound	actual
gigamax	6	25	30	27	27	12	12
eisenberg*	19	90	108	209	1512	38	540
abp	11	50	60	77	188	22	38
bakery*	58	285	342	1769	55039	116	424
treearb4	24	115	138	324	3675	48	464
elev23	1	0	0	2	2	2	2
coherence1	5	20	24	20	72	10	46
coherence2	14	65	78	119	3058	28	332
coherence3*	5	20	24	20	116	10	18
coherence4*	5	20	24	20	166	10	1370
elev33	1	0	0	2	2	2	2
elev43	1	0	0	2	2	2	2
tcp*	1	0	0	2	2	2	2

Table 4: Actual numbers of image computations performed by the minimization algorithms; for LY and BFH, the reported number is from the split routines, since this is what the bounds considered. The "lower bound" columns result from the bounds for LY, BFH, and PT by substituting the number of BR iterations for n, 1 for M, and 0 for all other variables. BR performs one image computation per iteration. LY shows 0 iterations if it terminated before executing the split loop, which means that no state in the good block could reach a state in the bad block.

attention to choosing splitters carefully.

7 Related Work

The algorithmic study of bisimulation goes back to Hopcroft, who proposed an $n \log n$ algorithm, where n is the number of states [27]. His algorithm handles only transition functions, rather than transition relations. Kanellakis and Smolka proposed an algorithm that handled transition relations, but did not match Hopcroft's lower bound [28]. The Paige-Tarjan algorithm improved on the Kanellakis-Smolka algorithm to match Hopcroft's bound [32]. Fernandez and Mounier presented an on-the-fly algorithm for deciding bisimulation equivalence [21]. All of these algorithms were defined for explicit-state representations. Bouali and de Simone proposed a BDD-based bisimulation algorithm [7]. Their algorithm computes the bisimulation relation (see Section 2), rather than the classes, so it is doubtful that it would scale to systems with more than a few million states (they reported handling systems with about four million states).

Several existing verification tools apply bisimulation minimizations to the components of a finite-state system description. Composition of these minimized components yields a finite-state system suitable for verification by model checking or other analyses. Bisimulation is also used to compare equivalence between finite-state transition systems. XEVE [6], a verification tool for an Esterel-based environment, appears to support both uses [8]. XEVE's underlying toolset, FC2TOOLS, provides minimization techniques for both explicit-state and symbolic representations, using the Kanellakis-Smolka and Bouali-de Simone algorithms, respectively [8].

The NCSU Concurrency Workbench [16] supports many forms of equivalence checking, including bisimulation equivalence. It is closely related to the Concurrency Factory project [14, 15], which supports bisimulation equivalence checking using techniques based on the Kanellakis-Smolka algorithm. The FDR analysis tool uses bisimulation minimization to provide a normal form for transition systems. and to minimize components before composition [34]. FDR does not support symbolic representations. Outside of a specific toolset, Aziz *et al.* take a compositional approach to bisimulation minimization in a symbolic setting [2]; Rahim's work takes a similar approach to verifying VHDL [33]. Both approaches also perform minimization relative to a given property. However, they represent equivalence relations (rather than classes).

	Split Marked			Don't Split Marked			Savings		
	Iter	Time	Mem	Iter	Time	Mem	Iter	Time	Mem
gigamax	271	9.2	5.66	6	2.0	5.58	98%	78%	1%
eisenberg*	587	16.5	4.67	270	9.3	4.28	54%	44%	8%
abp	134	6.3	3.96	19	2.0	3.86	85%	68%	3%
bakery*	2279	121.5	6.58	212	7.8	4.55	91%	94%	31%
treearb4	-	(24H)	(54)	232	118.3	6.06			
elev23	3	4.0	8.44	1	4.0	8.43	67%	0%	0%
coherence1	181	70.1	11.0	23	29.5	8.55	87%	58%	22%
coherence2	4376	2320.2	20.0	166	567.4	18	96%	76%	10%
coherence3*	3533	1003.8	18.0	9	7.9	7.89	100%	99%	56%
coherence4*	-	(33H)	(62)	685	13.8H	68			
elev33	3	7.3	11.0	1	7.2	11	67%	1%	0%
elev43	1	11.8	15.0	1	12.2	15	0%	-3%	0%
tcp*	1	3.2	7.84	1	3.2	7.83	0%	0%	0%

Table 5: The effects of not splitting marked blocks on the performance of the new PT algorithm. For each approach, the table shows the number of iterations required, the time taken (in seconds) and the memory used (in megabytes). The last set of columns shows the percentage of each resource saved when going to the optimized algorithm. A dash in the iterations column indicates that the algorithm did not complete in the time (in hours) indicated.

8 Conclusions

Bisimulation plays many roles in verification. It serves as a minimization technique for compositional or traditional model checking, as a means of checking equivalence between transition systems, or as a way of collapsing infinite-state systems into finite-state ones. Several theoretical analyses, experiments, and case studies attest to its efficiency and effectiveness in these areas [9, 17, 22, 28]. This paper concentrates on a particular usage: bisimulation minimization as a state-space reduction technique over the global state space in symbolic model checking.

Bisimulation minimization is attractive in the context of symbolic model checking because it is easy to compute symbolically and can be fully automated. Unfortunately, this approach does not appear worthwhile, as experience shows that the cost of minimizing the system outweighs the cost of model checking the unminimized system. One could argue that we don't see an improvement in resource usage because the minimization algorithms do not fundamentally alter the topology of the underlying BDDs, which is what generally gives rise to improvements in the performance of symbolic algorithms. However, we believe the real problem lies at an algorithmic, rather than a representational, level.

We have considered three bisimulation minimization algorithms: by Lee and Yannakakis, by Bouajjani, Fernandez, and Halbwachs, and by Paige and Tarjan. These algorithms mainly differ with respect to the extent to which they stabilize unreachable blocks. PT stabilizes all blocks, regardless of whether they are reachable. BFH stabilizes reachable blocks with respect to both reachable and unreachable blocks. As a result, it may stabilize some unreachable blocks that are split off from reachable blocks. LY stabilizes reachable blocks with respect to only reachable blocks. The differences between these techniques affect the numbers of operations of various types used in each algorithm and the number of iterations that each algorithm requires to terminate. From each algorithm, we created a novel on-the-fly model checker for invariant properties and compared it to model checking by backwards reachability.

Our analyses established close correlations between the sets of states computed during minimization and those computed during invariant property verification through backwards reachability. This implies that minimization and backwards reachability are similar in spirit in the context of testing invariant properties. Accordingly, we should not expect minimization to require fewer computational resources then model checking. However, each algorithm computes these sets in different ways; this prohibits us from claiming definitively that minimization algorithms require more resources than backwards reachability. We therefore provided two other analyses. The first compares the minimum number of operations of various kinds, such as image computations, across the four algorithms. The second is an experimental analysis on a suite of designs, such as controllers and protocols, that are typical in verification contexts. Both analyses support our claim that the costs of minimization outweigh those of model checking.

Combining all of this evidence, using bisimulation minimization either as a part of, or as a pre-processor to, model checking invariant properties of transition systems does not appear to be a viable approach. This does not imply that bisimulation has no role in verification contexts. Minimization in a compositional verification framework appears to make certain verification problems tractable that would not be so without minimization [2, 33]. Similarly, minimization can be used to collapse infinite-state systems into finite ones for purposes of exhaustive analyses [4, 24]. It remains to be seen what other implications our results have for the use of bisimulation in verification.

References

[1] Alur, R., C. Courcoubetis, D. Dill, N. Halbwachs and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 157–166, 1992.

[2] Aziz, A., V. Singhal, G. Swamy and R. Brayton. Minimizing interacting finite state machines: A compositional approach to language containment. In *Proceedings of the International Conference on Computer Design (ICCD)*, 1994.

[3] Beer, I., S. Ben-David, D. Geist, R. Gewirtzman and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193, Stanford, June 1994.

[4] Boreale, M. Symbolic bisimulation for timed processes. In *Proc. 5th Intl. Conference on Algebraic Methodology and Software TEchnology (AMAST)*, number 1101 in Lecture Notes in Computer Science, pages 321–335. Springer-Verlag, July 1996.

[5] Bouajjani, A., J.-C. Fernandez and N. Halbwachs. Minimal model generation. In Clarke, E. and R. Kurshan, editors, *Proc. Intl. Conference on Computer-Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 197–203. Springer-Verlag, 1990.

[6] Bouali, A. XEVE, an ESTEREL verification environment. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, pages 500–504, 1998.

[7] Bouali, A. and R. de Simone. Symbolic bisimulation minimization. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, number 663 in Lecture Notes in Computer Science, pages 96–108. Springer-Verlag, 1992.

[8] Bouali, A., A. Ressouche, V. Roy and R. de Simone. The FC2TOOLS set. In *Proc. of the 8th Intl. Conference on Computer Aided Verification (CAV)*, number 1102 in Lecture Notes in Computer Science, pages 441–445. Springer-Verlag, July/August 1996.

[9] Bourdellès, M. The steam boiler controller problem in ESTEREL and its verification by means of symbolic analysis. Technical Report 3285, INRIA Sophia Antipolis, October 1997.

[10] Clarke, E., R. Enders, T. Filkorn and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, August 1996.

[11] Clarke, E., O. Grumberg and D. Long. Model-checking and abstraction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[12] Clarke, E., M. Khaira and X. Zhao. Word level model checking – avoiding the Pentium FDIV error. In *Proc. 33rd ACM/IEEE Design Automation Conference*, pages 645–648, June 1996.

[13] Clarke, E. and R. Kurshan. Computer aided verification. *IEEE Spectrum*, 33:61–67, 1986.

[14] Cleaveland, R., J. Gada, P. Lewis, S. A. Smolka, O. Sokolsky and S. Zhang. The concurrency factory – practical tools for specification, simulation, verification, and implementation of concurrent systems. In *Proc. of DIMACS Workshop on Specification of Parallel Algorithms*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 75–90, May 1994.

[15] Cleaveland, R., J. Parrow and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):36–72, January 1993.

[16] Cleaveland, R. and S. Sims. The NCSU concurrency workbench. In *Proc. of the 8th Intl. Conference on Computer Aided Verification (CAV)*, number 1102 in Lecture Notes in Computer Science, pages 394–397. Springer-Verlag, July/August 1996.

[17] Creese, S. J. and A. Roscoe. TTP: A case study in combining induction and data independence. Unpublished draft manuscript, 1998.

[18] Dams, D. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technische Universiteit Eindhoven, 1996.

[19] Dams, D., O. Grumberg and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proc. 5th Int.l Conference on Computer-Aided Verification*, number 697 in Lecture Notes in Computer Science, pages 479–490. Springer-Verlag, 1993.

[20] Eiriksson, A. T. The formal design of 1M-gate ASICs. In Gopalakrishnan, G. and P. Windley, editors, *Proc. Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD)*, number 1522 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[21] Fernandez, J.-C. and L. Mounier. "On the fly" verification of behavioral equivalences and preorders. In *Proc. of the 3rd Intl. Conference on Computer Aided Verification (CAV)*, number 575 in Lecture Notes in Computer Science, pages 181–191. Springer-Verlag, 1991.

[22] Fisler, K. and M. Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In Gopalakrishnan, G. and P. Windley, editors, *Proc. Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD)*, number 1522 in Lecture Notes in Computer Science, pages 115–132. Springer-Verlag, 1998.

[23] Grumberg, O. and D. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[24] Hennessey, M. and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[25] Hennessy, M. and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM*, 32:137–161, 1985.

[26] Henzinger, T., O. Kupferman and S. Qadeer. From pre-historic to post-modern symbolic model checking. In *Computer Aided Verification, Proc. 10th Int. Conference*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[27] Hopcroft, J. E. An n log n algorithm for minimizing states in a finite automaton. In Kohavi, Z. and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[28] Kanellakis, P. C. and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[29] Kupferman, O. and M. Y. Vardi. On the complexity of branching modular model checking. In *Proc. 6th Conference on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 408–422, Philadelphia, August 1995. Springer-Verlag.

[30] Lee, D. and M. Yannakakis. Online minimization of transition systems. In *Proc. 24th ACM Symposium on Theory of Computing*, pages 264–274, Victoria, May 1992.

[31] Milner, R. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[32] Paige, R. and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16:973–989, 1987.

[33] Rahim, F. Property-dependent modular model checking application to VHDL with computational results. In *Proc. Third IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 1998.

[34] Roscoe, A. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice Hall, 1998.

[35] The VIS Group. VIS: A system for verification and synthesis. In Alur, R. and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer Verlag, July 1996.

A Proofs of Intermediate Lemmas

A.1 Lee-Yannakakis Lemmas

Proof of Lemma 5 Correlating the definition of BS_i and the variables in the LY code (Figure 5), on iteration i, B corresponds to BS_{i-1}, B' corresponds to BS_i, and B'' corresponds to $BS_{i-1} - BS_i$. By Lemma 4 then, on iteration i, $S_i = \overline{B'}$ and $F_i = B''$.

For the forwards direction, assume LY is in iteration i and goes to iteration $i+1$. Then $\langle B, init \rangle$ must have been enqueued on line 19 because only reachable blocks are ever on the queue and the algorithm stops if some block other than $\langle B, init \rangle$ becomes reachable. Since $\langle B, init \rangle$ was on the queue, $\emptyset \neq B \cap pre(B) \neq B$ or $\emptyset \neq B \cap pre(B'') \neq B$. Furthermore, LY did not terminate early at line 21, so $post(init) \cap B'' = \emptyset$. Thus, $post(init) \cap F_i = \emptyset$.

It therefore remains to show that $pre(F_i) \cap \overline{S_i} \neq \emptyset$. This is equivalent to showing that $pre(B'') \cap B' \neq \emptyset$ (by Lemma 4). Every state in B' was in B by definition (line 15). By definition of B (BS_{i-1}), every state in B has a transition to some state in B. By definition, $B = B' \cup B''$. Therefore, every state in B' either has a transition to a state in B' or to a state in B''. If some state has a transition to a state in B'', we are done. Therefore, assume that no state in B had a transition to B''. If this were true, then $B' \cap pre(B') = B'$. But if $B' \cap pre(B') = B'$ and $pre(B'') \cap B' = \emptyset$, then neither condition on line 19 is satisfied. However, one of the two conditions must have been satisfied since LY goes to iteration $i+1$. By contradiction, this direction therefore holds.

For the backwards direction, assume $pre(F_i) \cap \overline{S_i} \neq \emptyset$ and $post(init) \cap F_i = \emptyset$. Since $F_i = B''$, $post(init) \cap B'' = \emptyset$, so LY must not terminate early at line 21. Therefore, it remains to show that LY enqueues $\langle B, init \rangle$ at line 19. Since $post(init) \cap B'' = \emptyset$ and $init \in B'$, $B' \cap pre(B'') \neq B'$. We also assumed that $pre(F_i) \cap \overline{S_i} \neq \emptyset$. By the correlations given in the forwards direction and Lemma 4, $pre(B'') \cap B' \neq \emptyset$. Since $\emptyset \neq pre(B'') \cap B' \neq B'$, the second condition on line 19 is true, so $\langle B, init \rangle$ is enqueued. This direction therefore holds. □

Proof of Lemma 6 The proof is by induction on the number of iterations i taken through the split loop. For the base case, assume i is zero. LY starts iteration 1 of the split loop (reaches the split loop) if three conditions are met: (1) the initial state is a good state (line 4), (2) $post(init)$ does not reach a bad state (line 7), and (3) some good state reaches a bad state (line 8). BR goes to iteration 1 as long as the $init \notin F_0$ and F_0 is non-empty. The first LY condition guarantees that $init \notin F_0$. The third condition guarantees that some bad state must exist, hence F_0 is nonempty.

Assume the desired result holds for iteration $i-1$. We can therefore assume that both LY and BR entered iteration i. By Lemma 5, if LY is in iteration i and goes to iteration $i+1$, then $pre(F_i) \cap \overline{S_i} \neq \emptyset$ and $post(init) \cap F_i = \emptyset$. By definition, then $F_{i+1} \neq \emptyset$, so BR does not terminate on that condition. It therefore remains to show that $init \notin F_i$. This is equivalent to showing that $init$ is not in $pre(F_{i-1}) - S_i$. If $init$ had been in $pre(F_{i-1})$, though, LY would not have entered iteration $i+1$ (it would have terminated on line 21 in the previous iteration). Therefore, neither termination condition for BR can hold, so BR must also enter iteration $i+1$. □

A.2 Bouajjani-Fernandez-Halbwachs Lemmas

Proof of Lemma 11 The proof is by induction on the iterations through the BFH **while** loop (Figure 10, line 3). Initially, the only marked block is bad block. All new markings occur during the split loop. Assume BFH reaches the split loop and all states in marked blocks have paths to bad states. We must prove this still holds after executing the split loop. A block is marked under two conditions: if it is split from a marked block (lines 26 and 27), or if all states in it reach a marked block (lines 23 and 26). In the former case, since the block is a subset of a marked block, by the inductive hypothesis all states in the original block have a path to a bad state. The states in the new block therefore all have such a path. In the latter case, all states can reach a marked block, so all states have a path to a state that has a path to a bad state. The inductive case also holds, so the lemma holds. □

Proof of Lemma 12 The proof is by induction on i. In iteration 0, the good block is the only unmarked block. G_0 is the good block by definition, so the base case holds. Assume G_{i-1} is the only unmarked block in iteration $i-1$. We must prove that G_i is the only unmarked block in iteration i. By definition, $G_i = G_{i-1} \cap pre(G_{i-1}) - (pre(H_1) \cup \ldots \cup pre(H_{|\rho|-1}))$. Looking at the pseudocode in Figure 10, G_i is only marked if it is split off from a marked block (lines 26-27), or if it reaches a marked block (lines 23 and 26). Since G_{i-1} is not marked and G_i splits off from G_{i-1}, it does not get marked on lines 26 or 27. However, by definition, states in G_i reach only block G_{i-1}, which is unmarked by assumption. Figure 11 shows why G_i can be the only unmarked block. Every other new block contains states that reach at least one block other than G_{i-1} (because there is an intersection with $pre(H)$ for some block H other than G_{i-1} along every other path of the tree). Therefore, all other new blocks were marked in line 23 or line 26 of the pseudocode. G_i replaces block G_{i-1} (line 11), so G_i is the only unmarked block in iteration i. □

A.3 Paige-Tarjan Lemmas

Proof of Lemma 16 The proof is by induction on the number of iterations that have passed. In the base case, no iterations have been taken. In this case the desired result holds because the algorithm starts with the good block and the bad block, and the bad block is marked.

Assume the result is true after i iterations. Let B be the block chosen as the splitter in the $i + 1^{\text{st}}$ iteration. By the inductive hypothesis, there is at most one block D processed in the **foreach** loop starting on line 10. We prove that if the desired result holds before processing block D, then it holds after processing block D. While processing D, there are four candidate blocks created: D_1, D_2, D_{11}, and D_{12}. The latter two can only be created if D_2 is created. Furthermore, D_{11}, if created, replaces D_1.

There are two cases to consider, based on whether block B is marked. Table 2 summarizes when each block that could be created is marked in each case; the line numbers in the table indicate which lines in the algorithm mark the corresponding block. The desired result therefore holds if every column in the table preserves it. The first column creates new unmarked block D_2. However, block D is unmarked in this case and the new blocks replace D, so the inductive case holds. In the second column, since we assume there is at most one unmarked block before processing D, blocks D and B must be the same. At most one of the unmarked blocks (D_1 and D_{11}) exists at the end of the iteration, since D_{11} replaces D_1 if it is created. The inductive case holds in this case since the created blocks replace the single unmarked block D (equivalent to B). □

Proof of Lemma 17 The proof is by induction on the number of iterations taken through the **while** loop at line 5 (Figure 13). Initially, two blocks are created: the good block and the bad block. Only the bad block is marked (line 4). Every path in this block has a trivial path to itself, so the base case holds. All other block markings arise from the algorithm. It therefore suffices to consider how the algorithm handles each newly created block D_1, D_2, D_{11}, and D_{12}. The decision whether to mark each blocks occurs on lines 15, 18, 25, and 29 of the code in Figure 13, respectively. We consider each in turn.

- D_1 is marked (line 15) iff B is marked. By definition, every state in D_1 reaches B. If B is marked, then every state in D_1 must have a path to a bad state by going through block B. D_1 is therefore marked correctly.
- D_2 is marked (line 18) iff B is not marked. By definition (line 12), no state in D_2 reaches B. If B is not marked, then all other blocks must be marked by Lemma 16. Since the transition is total, every state in D_2 must therefore have a path to a bad state. D_2 is therefore marked correctly.
- D_{11} is marked (line 25) iff D_1 is marked or B is marked. By definition (line 20), D_{11} is a subset of D_1, so if D_1 is marked, D_{11} should also be marked. Furthermore, if B is marked, D_1 would have been marked if B was marked. D_{11} is therefore marked correctly.
- D_{12} is automatically marked (line 29). By definition (line 21), states in D_{12} reach both block B and blocks other than B. Since there is at most one unmarked block by Lemma 16, each state in D_{12} must therefore reach some marked block, so it is correctly marked.

□

Proof of Lemma 18 By definition, U_i is stable with respect to B and $S - B$ iff for each set, either all states in U_i reach the set or none of them do. U_i is one of the blocks D_1, D_2, D_{11}, or D_{12}, as defined in the pseudocode in Figure 13. E_1 (line 9) is the set of all states that reach B. By definition, D_1 is stable with respect to B because all elements of D_1 reach B. Since D_{11} and D_{12} are subsets of D_1, they are also stable with respect to B. D_2 is also stable with respect to B because no state in D_2 reaches B by definition.

E_2 (line 10) is the set of all states that reach B, but that don't reach $S - B$. D_{11} is clearly stable with respect to $S - B$ since all states in D_{11} are in E_2. D_{12} is stable with respect to $S - B$ because all of its elements reach $S - B$. If one of D_{11} or D_{12} is empty, then either everything or nothing in D_1 is in E_2, so D_1 is stable with respect to $S - B$.

Finally, D_2 must be stable with respect to $S - B$ because the transition relation is total: all states in D_2, not reaching states in B, must reach states in $S - B$. Since each of D_1, D_2, D_{11}, and D_{12} is stable with respect to B and $S - B$, and since U_i is one of these blocks, U_i is stable with respect to B and $S - B$. □

Proof of Lemma 19 The proof is by induction on i. At the first iteration, there are only two blocks. U_1 is the good block and M is the bad block. They are both in the same block of X, so the base case holds trivially.

For the inductive case, assume that U_i is stable with respect to all blocks in other blocks of X at iteration i. Let $M_1, \ldots, M_k$ be the other blocks in the same block of X as U_i. We must prove that at iteration $i + 1$, U_{i+1} is stable with respect to all blocks in other compound blocks of X. The PT algorithm never merges blocks of X. Furthermore, U_{i+1} is a subset of U_i and sub-blocks inherit stability from the blocks they split off from. Therefore, the only blocks U_{i+1} could be unstable with respect to are $M_1, \ldots, M_k$. There are now three cases to consider, depending upon which block is chosen as the splitter in iteration i.

1. U_i is the splitter, U_{i+1} is D_{11}, and U_i goes into its own block at line 8 and $M_1, \ldots, M_k$ stay in block $S - B$. Let M_j be one of these blocks. If U_{i+1} were unstable with respect to M_j, then some elements of U_{i+1} would reach M_j and others would not (by the definition of stability). But by definition of D_{11} from U_i, D_{11} does not reach a state in $S - B$. U_{i+1} is therefore stable with respect to M_j.

2. Some M_j from the same compound block as U_i is chosen as the splitter. Then $B = M_j$ goes into its own block. U_{i+1} is stable with respect to B at the end of the iteration by Lemma 18. The remaining M blocks remain in the same compound block as U_i for the next iteration. The desired result therefore holds in this case.

3. Some other block that is not in the same compound block as U_i is chosen as the splitter. By the inductive hypothesis, U_i is stable with respect to this splitter, so the block of X containing U_i is unchanged during this iteration. The desired result therefore holds by the inductive hypothesis.

□

Proof of Lemma 20 There are two cases, depending upon whether U_i is chosen as the splitter in iteration i. If it is, then by construction (line 9) $U_{i+1} \subseteq U_i \cap pre(U_i)$, so $U_{i+1} \subseteq pre(U_i) \subseteq pre(U_1 \cup M_1 \cup \ldots \cup M_k)$. If U_i is not the splitter, then under the given assumptions, the splitter is some M_j. By definition, $U_{i+1} = D_2 = U_i - pre(M_j)$. By construction (line 13), each state in D_2 reaches something in $S - B$; in this case $S - B$ is $U_i, M_1, \ldots, M_{j-1}, M_{j+1}, \ldots, M_k$. Therefore, $U_{i+1} \subseteq pre(U_i \cup M_1 \cup \ldots \cup M_{j-1} \cup M_{j+1} \cup \ldots \cup M_k)$. This completes the proof. □

Proof of Lemma 21 The proof is by induction on i. In the base case, i is 1. At the start of the first iteration, X has only the single block $\{good, bad\}$. Hence, $M_1 = bad$, which by definition is equivalent to, and thereby contained in, F_0.

For the inductive case, let $\{U_i, N_1, \ldots, N_j\}$ be the block of X containing U_i at the start of iteration i. Assume $N_1 \cup \ldots \cup N_j \subseteq F_0 \cup \ldots \cup F_{i-1}$. Let $\{U_{i+1}, M_1, \ldots, M_k\}$ be the block of X that contains U_{i+1} at the start of iteration $i + 1$. We must prove that $M_1 \cup \ldots \cup M_k \subseteq F_0 \cup \ldots \cup F_i$.

The definition of the M blocks depends on which block was chosen as the splitter in iteration i. There are three cases:

1. U_i is the splitter. Then at line 8 (Figure 13), U_i goes into its own block. The M blocks will therefore consist of those blocks other than U_{i+1} that are created while splitting U_i. Since U_i is the splitter, U_{i+1} is either D_{11} or D_1 from Figure 13, depending upon which one is defined. The M blocks therefore consist of D_2 and possibly D_{12} (the latter may not be defined). D_{11} and D_{12} partition D_1. Since we work only with unions in the rest of the proof, we finish the proof with respect to D_{11} and D_{12} instead of creating separate cases for each possible combination of non-empty blocks.

   Since the M blocks are D_2 and D_{12}, we must show that $D_2 \cup D_{12} \subseteq F_0 \cup \ldots \cup F_i$. By definition, $D_2 \cup D_{12}$ is $U_i \cap pre(S - U_i)$, where S is the compound block $\{U_i, N_1, \ldots, N_j\}$. Therefore, $D_2 \cup D_{12}$ is $U_i \cap pre(N_1 \cup \ldots \cup N_j)$. By the inductive hypothesis, $N_1 \cup \ldots \cup N_j \subseteq F_0 \cup \ldots \cup F_{i-1}$. So $pre(N_1 \cup \ldots \cup N_j) \subseteq pre(F_0 \cup \ldots \cup F_{i-1})$. But $pre(F_0 \cup \ldots \cup F_{i-1}) \subseteq F_0 \cup \ldots \cup F_i$ by Lemma 2, so $D_2 \cup D_{12} \subseteq F_0 \cup \ldots \cup F_i$, which completes the proof for this case.

2. One of the N blocks is the splitter. Let N_m be the chosen splitter. Then at line 8 (Figure 13), N_m goes into its own block and $S-B$ is the block $\{U_i, N_1, \ldots, N_{m-1}, N_{m+1}, \ldots, N_j\}$. U_{i+1} is defined to be D_2. In the process of splitting, PT adds blocks D_{11} and D_{12} to $S-B$. Therefore, the M blocks are $D_{11}, D_{12}, N_1, \ldots, N_{m-1}, N_{m+1}, \ldots, N_j$. By the inductive hypothesis, each N block is in $F_0 \cup \ldots \cup F_{i-1}$. Therefore, it remains to show that $D_{11} \cup D_{12} \subseteq F_0 \cup \ldots \cup F_i$.

   By definition, since the splitter (B) is N_m, $D_{11} \cup D_{12}$ is $U_i \cap pre(N_m)$. By the inductive hypothesis, N_m is contained in $F_0 \cup \ldots \cup F_{i-1}$. Therefore, $pre(N_m)$ is contained in $pre(F_0 \cup \ldots \cup F_{i-1})$. By Lemma 2, $pre(F_0 \cup \ldots \cup F_{i-1})$ is contained in $F_0 \cup \ldots \cup F_i$. Therefore, $D_{11} \cup D_{12} \subseteq F_0 \cup \ldots \cup F_i$, which completes the proof for this case.

3. The splitter is some block from a different compound block than U_i. Since U_i is stable with respect to this splitter by Lemma 19, the M blocks are identical to the N blocks. Since the N blocks are contained in $F_0 \cup \ldots \cup F_{i-1}$, the M blocks are contained in $F_0 \cup \ldots \cup F_i$.

□

Proof of Lemma 22 The proof is by induction on i. In the base case, i is zero. By definition, S_0 is the set of bad states and U_0 is the set of good states. $\overline{S_0} = good$, so $\overline{S_0} \subseteq U_0$. For the inductive case, assume $\overline{S_i} \subseteq U_i$. We must prove that $\overline{S_{i+1}} \subseteq U_{i+1}$. Let x be some element from $\overline{S_{i+1}}$. We will prove that x must be in U_{i+1}.

By definition, $S_{i+1} = S_i \cup pre(F_i)$. Therefore, $x \notin S_i$ and $x \notin pre(F_i)$. Since $x \notin S_i$, $x \notin pre(F_0 \cup \ldots \cup F_{i-1})$ by Lemma 1. Extending this with $x \notin pre(F_i)$ yields $x \notin pre(F_0 \cup \ldots \cup F_i)$. Let $\{U_i, M_1, \ldots, M_k\}$ be the block of X that contains U_i at the end of iteration i. Since $x \notin pre(F_0 \cup \ldots \cup F_i)$, then by Lemma 21, $x \notin pre(M_1 \cup \ldots \cup M_k)$. Furthermore, since $x \in U_i$ and $x \notin pre(M_1 \cup \ldots \cup M_k)$, $x \in pre(U_{i-1})$ by Lemma 20. By construction, $M_1, \ldots, M_k$ includes those blocks other than U_i that split off from U_{i-1}. Since x does not reach any of the M blocks, it must be in $pre(U_i)$.

There are now three cases to consider depending upon which block was chosen as the splitter in iteration i.

1. U_i was the splitter. Then U_{i+1} is D_{11} or D_1 (depending upon which is defined). Since D_{11} is a subset of D_1, we complete the proof using the definition of D_{11}. $D_{11} = U_i \cap pre(U_i) - pre(M_1 \cup \ldots \cup M_k)$. x is in U_i by the inductive hypothesis since it is in $\overline{S_i}$. We concluded earlier that x is in $pre(U_i)$. We also concluded that x is not in $pre(M_1 \cup \ldots \cup M_k)$. Therefore, x is in U_{i+1}.

2. One of the M blocks, M_j, was the splitter. Then by definition, U_{i+1} is $D_2 = U_i - pre(M_j)$. By the inductive hypothesis, x is in U_i since it is in $\overline{S_i}$. Since $x \notin pre(M_1 \cup \ldots \cup M_k)$, $x \notin pre(M_j)$. Therefore, x must be in U_{i+1}.

3. Some block in a different block of X than U_i was the splitter. By Lemma 19, U_i is stable with respect to the splitter. Therefore, $U_{i+1} = U_i$. Since $S_i \subseteq S_{i+1}$ by definition, $\overline{S_{i+1}} \subseteq \overline{S_i}$. Element x is therefore in $\overline{S_i}$, and also in $U_i = U_{i+1}$ by the inductive hypothesis.

□

Proof of Lemma 23 By definition, $F_i = pre(F_{i-1}) - S_{i-1}$. Therefore, $F_i \subseteq \overline{S_{i-1}}$, which makes $F_i \subseteq U_{i-1}$ by Lemma 22. If U_{i-1} is stable, then by definition all elements in U_{i-1} agree on their transitions to other blocks. By definition, each state in F_i has a path to a bad state. Therefore, if all states in U_{i-1} have transitions to states in U_{i-1}, then the invariant property fails because each state must have a path to a bad state. If no state in U_{i-1} has a transition to U_{i-1}, then since the transition relation in total, each state must have a transition to some block other than U_{i-1}. However, that other block must be marked, so every state in U_{i-1} has a path to a bad state. The invariant property also fails in this case. □