Bisimulation Minimization in an Automata-Theoretic Verification Framework*

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}

October 27, 1998

Abstract

Bisimulation is a seemingly attractive state-space minimization technique because it can be computed automatically and yields the smallest model preserving all μ-calculus formulas. It is considered impractical for symbolic model checking, however, because the required BDDs are prohibitively large for most designs. We revisit bisimulation minimization, this time in an automata-theoretic framework. Bisimulation has potential in this framework because after intersecting the design with the negation of the property, minimization can ignore most of the atomic propositions. We compute bisimulation using an algorithm due to Lee and Yannakakis that represents bisimulation relations by their equivalence classes and only explores reachable classes. This greatly improves on the time and memory usage of naïve algorithms. We demonstrate that bisimulation is practical for many designs within the automata-theoretic framework. In most cases, however, the cost of performing this reduction still outweighs that of conventional model checking.

1 Introduction

Symbolic model checking has had a dramatic impact on formal verification [4, 11]. Binary decision diagrams (BDDs) provide an efficient and canonical symbolic representation for digital systems, which enables a complete analysis of systems with exceedingly large state-spaces. BDD-based model checkers have enjoyed substantial and growing industrial use over the last few years, discovering very subtle flaws in designs of considerable complexity. Nevertheless, the growing size of current and future semiconductor designs seriously challenges current model checking technology. Despite technological advances, the unremitting increase in design complexity, known as the *state-explosion* problem, remains a serious obstacle to industrial-scale verification.

Research into combating state explosion takes many forms, including alternate representations of transition systems [3, 32], compositional reasoning [20, 27], and various state-space reduction techniques [10, 14, 25]. Most of the latter techniques are a form of *abstraction.* An abstraction suppresses information from a concrete state-space by mapping it into a smaller, abstract state-space. Each set of concrete states mapping to a single abstract state forms an *equivalence class* under the abstraction. Hopefully, the abstract model is more amenable to model checking. The abstraction must, however, be *safe*: verification in the abstract model must "carry over" to the concrete model. More formally, the abstraction must preserve the truth

*A shorter version of this paper appeared in the proceedings of the International Conference on Formal Methods in Computer-Aided Design [17].

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

or falsehood of the properties of interest. Abstractions preserving both truth and falsehood provide *strong preservation*, while those preserving only truth provide *weak preservation*.

Ideally, we want abstractions to meet two conditions. First, they should yield the smallest safe abstract models. Second, constructing them should require minimal manual involvement. The second is crucial for abstraction to work well in an industrial environment. Current techniques generally fail to satisfy this criterion, which significantly limits their applicability. For the first, we need a means to compare the size of abstract models. We say that abstraction A_1 is *coarser* than abstraction A_2 if A_1 collapses some subsets of A_2's equivalence classes together. The coarser the abstraction, the smaller the abstract state space. The properties to be preserved generally dictate the maximal coarseness of safe abstractions.

Bisimulation [31] is a general abstraction satisfying both these conditions. Bisimulation collapses states that have both the same local properties and bisimilar successor states. It is the coarsest abstraction that strongly preserves all properties expressible in the propositional μ-calculus, which expressively subsumes CTL* (an extension of both the linear temporal logic LTL and the branching temporal logic CTL) [23]. Furthermore, it can be computed in a completely automated fashion, requiring no understanding of the design under consideration. It is straightforward to compute symbolically, and therefore suggests a good approach to state-space reduction in the context of symbolic model checking. Unfortunately, as reported by Clarke *et al.* [9], the BDDs used to compute the bisimulation relation get overly large in practice; accordingly, they concluded that computing bisimulations is not a feasible approach to algorithmic state-space reduction.

This paper revisits bisimulation reductions, this time in an automata-theoretic framework [28, 33]. The change of framework is significant. In an automata-based framework, model checking involves intersecting the design automaton with an automaton representing the negation of the property, then checking whether a fair path exists. Restricted to this single (and simple) property, the bisimulation abstraction is even coarser than bisimulation on the design alone because the atomic propositions of the design need not be considered.

Performed naïvely, this approach still suffers from the problem of "exploding" BDDs because the bisimulation relation is a binary relation on the state space (thus, if the states are described in terms of n Boolean variables, the bisimulation relation is over $2n$ Boolean variables, resulting in very large BDDs). We overcome this problem by using an algorithm due to Lee and Yannakakis [29]. This algorithm has two key ideas: first, it computes bisimulations only on the reachable state space; second, it computes the equivalence classes of the bisimulation relation rather than the bisimulation relation itself. By combining the automata-theoretic approach with the Lee-Yannakakis algorithm, we are able to abstract much larger designs using bisimulation than was previously possible. Unfortunately, while computing our abstraction can result in significant reduction in the size of the state space, the gain obtained by the reduction is often offset by the cost of computing it. Thus, state-space reduction via bisimulation abstractions is a useful technique, but its applicability is not universal.

The outline of the paper is as follows. Section 2 discusses our experimental framework. Section 3 reviews bisimulation and the algorithms used for computing it. Section 4 describes our experiences using the Lee-Yannakakis algorithm in the automata-theoretic framework. We discuss a variant of the algorithm in Section 5 and comparative results with traditional model checking in Section 6. Section 7 presents related work. Concluding remarks and future directions appear in Section 8.

2 Experimental Framework

We conducted our experiments in a framework built upon the VIS model checker [19]. We chose VIS for several reasons: first, it supports automata-theoretic model checking; second, as a tool, it is engineered to make extensions relatively easy; third, it uses Verilog as an input language, for which examples are easier to find than for other model checker input languages.

Our experiment suite contains nine designs, most taken from the VIS distribution. They include a 3-bit counter, a traffic-light controller, an arbiter, two mutual exclusion protocols, a cache coherence protocol, a railway controller, an elevator, and a tree-structured arbiter. The tested properties are taken from the properties distributed with the examples. Whenever possible, we test both a safety property and a liveness property of each design. Our experiments also include properties that do not hold of their respective designs. We manually translate the negated properties into Verilog state machines. For each experiment described, we use VIS as a front-end to get a symbolic representation of the corresponding transition system.

Our algorithms work on these symbolic representations, using operations provided by VIS for reachability computations. In an attempt to control the intermediate BDD sizes, we use VIS's partitioned transition relation representation for the original transition system. The minimized transition system has a monolithic transition relation. All experiments were run on an UltraSparc 140 with 128 megabytes of memory running Solaris.

One important note regarding memory statistics is in order. Within VIS, we used Somenzi's CUDD package [13]. Depending upon the setting of the `LooseUpTo` parameter, CUDD may use substantially more memory than is actually needed for a computation (as much as a factor of two in our experience) in an attempt to improve time efficiency. We have set this parameter to a low value (100) for our runs to make our memory statistics as tight as possible. As these figures may still be imprecise, we also report the maximum number of live BDD nodes during each computation. CUDD does not provide this figure, but we have approximated it manually by inserting checks throughout our code.

3 Computing Bisimulation

Bisimulation is an equivalence relation on the states of a transition system [31]. It distinguishes states based on two requirements: whether they agree on the values of a given subset of atomic propositions, and whether transitions from each state lead to bisimilar states. The subset of atomic propositions on which states must agree depends on the application. A general bisimulation-based minimization would consider all variables (both state variables and atomic propositions) that could be referenced in properties. When verifying a particular property, however, only those state variables and atomic propositions affecting the property need to be considered [31]. Our experiments heed this observation.

Formally, we define transitions systems and bisimulation as follows. A *transition system* is a tuple $\langle S, R, AP, L, F\rangle$, where S is the set of states, $R \subseteq S \times S$ is the transition relation, AP is the set of atomic propositions, $L \subseteq S \times AP$ indicates which atomic propositions are true in each state, and $F \subseteq 2^{AP}$ captures the fairness condition. Each element of F indicates a set of states in which a single fairness constraint holds. We use Büchi fairness constraints, so fair paths pass through a state of each element of F infinitely often. The sets of states comprising F may be specified at the user level using formulas. For purposes of discussion, however, we prefer the state view. We assume that there is a unique atomic proposition ap_{fi} corresponding to each set F_i in F such that for all states s in S, (s, ap_{fi}) is in L iff s is contained in F_i. These atomic propositions are used to ensure that the fairness condition is preserved under bisimulation.

Given a transition relation $\langle S, R, AP, L, F\rangle$ and a subset AP' of AP, a binary relation $B \subseteq S \times S$ is a *bisimulation relation relative to* AP' iff the following conditions hold for every pair of states (s_1, s_2) in B:

1. For every atomic proposition ap in AP', (s_1, ap) is in L iff (s_2, ap) is in L.
2. For all s_1' such that (s_1, s_1') is in R there exists s_2' such that (s_2, s_2') is in R and (s_1', s_2') is in B.
3. For all s_2' such that (s_2, s_2') is in R there exists s_1' such that (s_1, s_1') is in R and (s_1', s_2') is in B.

We can compute bisimulation symbolically by computing its complement, *i.e.*, those pairs of states that fail to meet one of the above three conditions. We are interested in the maximum bisimulation relation: that which distinguishes as few states as possible. Computing the least fixpoint of the following equation yields the complemented maximum relation:

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

In our experiments, this bisimulation computation completed only for the smaller examples, as shown in Table 1.[1] These results strongly indicate that plain symbolic bisimulation computation is an infeasible state-space reduction technique for large designs and support the similar conclusion drawn by Clarke *et*

[1] These runs used Cudd's default `LooseUpTo` value of 800000 in order to save computation time. The eisenberg computations each took over a full day to complete.

	State Variables	Atomic Propositions	Equivalence Classes	Max Live Nodes	Memory (MB)
counter	3	1	8	127	2.68
tlc-1	7	2	26	991	2.86
tlc-2	7	3	38	1184	2.80
k-elev	10	3	246	15576	3.4
eisenberg-1	17	2	9804	422642	34
eisenberg-2	17	2	10100	424757	20

Table 1: Computing bisimulation with the symbolic algorithm. The numbers after the design names distinguish the various properties under consideration. The types of each property appear in Table 3. The "Atomic Propositions" column indicates how many variables appear in the respective property.

al. [9]. Section 4 shows that bisimulation becomes more feasible for large designs under three conditions: (1) when we represent bisimulation by its equivalence classes rather than the binary relation, (2) when we consider only the reachable equivalence classes while performing bisimulation, and (3) when we minimize after intersecting the automaton for the negation of the desired property.

4 Making Bisimulation More Tractable

Bisimulation as presented in the previous section becomes intractable largely because the computed relation is defined over $2n$ variables, where n is the number of boolean state variables in the original design. This doubling yields extremely large BDDs even for moderate-size designs. In addition, the relation may distinguish large numbers of states by considering many atomic propositions in the original design. Although in general large relations do not necessarily imply large BDDs, our evidence shows that the bisimulation relations are large in practice. Reducing the size of the relation itself is a reasonable approach to try to make the resulting BDDs more tractable.

Our experiments address both problems. For the former, we use an algorithm due to Lee and Yannakakis [29] that represents the bisimulation relation by its set of equivalence classes (each defined over n variables), rather than as a monolithic binary relation (over $2n$ variables). Section 4.1 provides an overview of the algorithm. For the latter, we intersect the negation of the property with the design before minimization, thereby reducing the number of atomic propositions that need to be considered. The remaining parts of this section show that running the algorithm on the intersected automaton does render bisimulation more tractable as a minimization technique.

4.1 The Lee-Yannakakis Algorithm

The Lee-Yannakakis bisimulation algorithm has two desirable features. First, it represents the relation by its equivalence classes, instead of by its pairs of related states. Second, it only computes bisimulation on the reachable state space. Model checkers have long ignored unreachable states for sake of efficiency. The traditional formalization of bisimulation, however, does not distinguish between equivalence classes that contain reachable states and those that do not. As our experiments show, the number of equivalence classes containing no reachable states can be substantial. This parallels the large number of unreachable states in most systems.

The combination of minimization and reachability offers two further advantages. First, the algorithm can minimize on-line. By augmenting the algorithm with an on-the-fly model checker, it is conceivable to verify designs which are too large for straightforward model checking, or for traditional minimization techniques. Second, the combination allows representatives to be selected such that if there is an edge between two blocks in the minimized system, then there is an edge between their representatives in the original system. This is useful for reporting error traces during model checking. Any path through the minimized system corresponds to a legitimate path through the original design by following the representatives.

begin
    stack := $\{\langle B, i\rangle\}$, where i is the initial state and B is the block containing i;
    queue := $\emptyset$;
    partition contains one block per combination of atomic propositions ;

search:
    while stack $\neq \emptyset$ do **begin**
        $\langle B, p\rangle :=$ pop(stack) ;
        $D := R(B)$;
        foreach block $\langle C, q\rangle$ containing a state in $R(p)$ **do begin**
            if B contains a state with no successor in C **then** enqueue $\langle B, p\rangle$;
            if q is not defined **then** select p_c in $C \cap R(p)$ and set $q := p_c$;
            add edge $\langle B, p\rangle \to \langle C, q\rangle$;
            $D := D - C$;
        end ;
        if $D \neq \emptyset$ **then** enqueue $\langle B, p\rangle$;
    end

split:
    while queue $\neq \emptyset$ **do begin**
        $\langle B, p\rangle :=$ delete (queue) ;
        $B' := \{q \in B : \text{blocks}(R(q)) = \text{blocks}(R(p))\}$;
        $B'' := B - B'$;
        $B := B'$;
        add a block with stateset B'' to the partition ;
        foreach edge $\langle C, q\rangle \to \langle B, p\rangle$ in the minimized system **do begin**
            if some elements of C reach B and some do not **then** enqueue $\langle C, q\rangle$;
            if some elements of C reach B'' and some do not **then** enqueue $\langle C, q\rangle$;
            if no successor or q is in B **then** delete edge $\langle C, q\rangle \to \langle B, p\rangle$;
            if some successor of q is in B'' **then**
                if the block for B'' is not marked **then**
                    select p''_b in $R(q) \cap B''$, mark, and push block onto stack ;
            add edge $\langle C, q\rangle \to \langle B'', p''_b\rangle$;
        end
        if stack $\neq \emptyset$ **then goto** search ;
    end

Figure 1: The Lee-Yannakakis combined bisimulation and reachability algorithm [29]. R is the transition relation of the system being minimized. The term "block" refers to an equivalence class and its representative. In this pseudocode, the *enqueue* operation checks whether a block is already in the queue before enqueuing it.

	State Variables	Reachable States	Reachable Classes	% Reach. Saved	Max Live Nodes	Memory (MB)
counter	3	8	8	0%	45	2.7
tlc-1	7	20	13	35%	392	2.87
tlc-2	7	20	20	0%	439	2.87
k-elev	10	262	132	50%	2436	2.99
eisenberg-1	17	1611	1214	25%	28810	4.56
eisenberg-2	17	1611	1076	33%	28859	4.58
bakery-1	20	2886	2546	18%	68543	5.47
arbiter	23	73	64	12%	8477	3.29
treearbiter4-1	23	5568	288	95%	24129	4.7
treearbiter4-2	23	5568	1344	76%	58805	5.82
dcnew-1	24	186876	230	100%	10870	5.94
dcnew-2	24	186876	235	100%	10303	5.94

Table 2: Computing bisimulation plus reachability with the Lee-Yannakakis algorithm. The properties for the new designs (bakery, arbiter, dcnew, and treearbiter) each referred to two atomic propositions. The figures for bakery-1 also hold for bakery-2 as both properties use the same atomic propositions. The "% Reach. Saved" column reflects the ratio of reachable classes to reachable states.

The algorithm starts with a partition of the states into equivalence classes based only on the atomic propositions. Each reachable equivalence class also stores a single reachable state acting as the representative of the class. Our implementation represents each of these with a BDD. Two structures, a stack and a queue, store equivalence classes that need further processing. Classes on the stack must be processed to construct their outgoing edges. Classes in the queue must be split into multiple equivalence classes in order to preserve the bisimulation relation. The algorithm terminates when both the stack and the queue are empty. As the classes are represented explicitly (even though their elements are represented symbolically), this algorithm can be viewed as converting a symbolically-represented transition system into an explicitly-represented, minimized transition system. A summary appears in Figure 1.

4.2 Minimized Bisimulation on Designs

Our first experiment used the Lee-Yannakakis algorithm to compute bisimulation on the designs alone (without intersecting the negated property). Each minimization considered only the atomic propositions appearing in the respective property and those for the fairness constraints. As shown in Table 2, representing the equivalence classes and accounting for reachable classes does enable us to compute bisimulation more efficiently and on a larger collection of designs. Comparing the number of reachable classes to the total number of equivalence classes shown in Table 1, we see that the percentage of classes that are reachable can be rather small on larger examples. Based on these figures, representation and reachability appear to be significant factors in bisimulation's tractability.

4.3 Minimizing Relative to Properties

The bisimulation computations in the previous section considered only those atomic propositions appearing in properties or required to support fairness constraints. However, it is possible to further reduce the number of atomic propositions under consideration. As we are working within an automata-theoretic framework, once the negation of the property has been intersected with the original design automaton, the property to be checked reduces to "does there exist a fair path" ($\mathsf{EG}\mathit{true}$ in CTL). Algorithmically, we customize bisimulation minimization to this property at the level of the atomic propositions. Rather than consider the atomic propositions in the property, we now use only those corresponding to the fairness constraints. Specializing to the property therefore does not require any change to the Lee-Yannakakis algorithm or any additional machinery.

	Property	Reach. States	Reach. Classes	% Reach. Saved	Max Live Nodes	Memory (MB)
counter	GFp	23	16	30%	139	2.77
tlc-1	Gp	20	13	35%	472	2.91
tlc-2	G(p → Fq)	46	30	35%	649	2.95
k-elev	G(p → Fq)	648	266	59%	5035	3.26
eisenberg-1*	Gp	3198	2074	35%	50447	5.39
eisenberg-2	G(p → Fq)	4155	2713	35%	53580	5.72
bakery-1*	G(p → Fq)	8079	7258	10%	118871	8.04
bakery-2	G(p → Fq)	5822	5302	9%	103102	7.4
arbiter	G(p → Fq)	185	158	15%	21518	3.86
treearbiter4-1	Gp	5568	1	100%	19938	6.54
treearbiter4-2	G(p → Fq)	13392	3320	75%	162212	12
dcnew-1	Fp	186876	1	100%	2286	4.03
dcnew-2*	GFp	445404	242	100%	15579	4.5
coherence	Gp	94738	1	100%	18099	7.73

Table 3: Computing bisimulation on the DesignProp automaton. Variables p and q in the properties may refer to boolean combinations of atomic propositions; they are independent across properties. Asterisks indicate properties that fail to hold of their respective designs. The newly handled design, coherence, has 37 state variables (before intersection with its property). The coherence property refers to four atomic propositions.

We refer to the intersection of the design and the negated property as the *DesignProp automaton*. Table 3 shows the reductions achieved by computing bisimulation on these automata. As the table shows, we are now able to minimize one design (coherence) that were not amenable to previous computations. The numbers of reachable states and reachable classes are now relative to the DesignProp automaton, rather than to the original design as in Table 2. The transition system for DesignProp has additional state variables as compared to the design. The number of additional variables depends on the type of property involved. A safety property contributes one state variable, while the liveness properties considered here contribute two.

These extra variables account for the additional states and any increased numbers of live nodes needed for the computation. The numbers of live nodes are generally much higher for computing bisimulation on the DesignProp automaton, often roughly 50% higher. For treearbiter4-1, however, the maximum number of live nodes actually decreases. We attribute this to the small number of reachable classes (in this case, only one) in the DesignProp automaton.

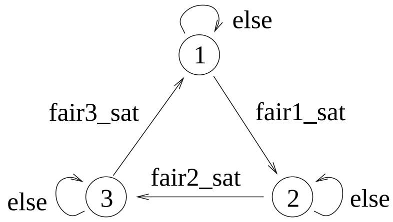
Given that we can now handle more designs, minimizing the DesignProp automaton seems promising. From the perspective of the Lee-Yannakakis algorithm, we did nothing more than reduce the number of atomic propositions considered in forming the initial partition. Given the definition of bisimulation presented in Section 3, it is not surprising that reducing the number of atomic propositions used can greatly impact the degree of minimization achieved, because the atomic propositions dictate the initial partitioning into equivalence classes. Accordingly, further reducing the number of required atomic propositions might prove even more promising. The next section discusses an experiment that requires only one atomic proposition per design, regardless of the number of fairness constraints.

4.4 Minimizing with One Atomic Proposition

The experiments reported in the previous section considered one atomic proposition per fairness constraint. Further reducing the number of atomic propositions therefore requires reducing the number of fairness constraints. The fairness constraints indicate sets of states that must be passed through infinitely often. Any method that satisfies the constraints is sufficient, as long as it requires each set to be visited infinitely often. One such method checks the fairness constraints in some linear order [8], as shown in the following diagram.

	Num Fair	Reach. States	Reach. Classes	% Reach. Saved	% Classes Saved	Max Nodes	Memory (MB)
tlc-1	2	28	1	96%	92%	554	2.98
tlc-2	2	56	10	82%	67%	717	3.03
k-elev	4	3664	128	97%	52%	12349	3.96
eisenberg-1*	6	10039	3590	64%	-73%	113354	11
eisenberg-2	6	16228	2096	87%	23%	81224	7.1
bakery-1*	4	25081	5297	79%	27%	146466	9.26
bakery-2	4	21815	1417	94%	73%	78743	6.19
arbiter	3	353	97	73%	39%	55937	5.55
treearbiter4-2	4	26712	2365	91%	29%	461598	38

Table 4: Single fairness statistics. The "Num Fair" column indicates the number of constraints of the original design; it does not account for the fairness constraint added by the property automaton. The "% Classes Saved" column reflects the ratio of reachable classes in the OneFair automaton to the reachable classes in the DesignProp automaton (Table 3). Designs not covered in this table had no fairness constraints. We did not produce a OneFair automaton for treearbiter4-1 because its DesignProp automaton already yields only one reachable block.



This diagram shows the linearization of three fairness constraints using an automaton. The automaton has one state for each fairness constraint. Each state has two transitions: one going to the next state in the ordering when the associated fairness constraint is satisfied, and one looping back to the same state for when the constraint is not satisfied. If the fairness constraint for this automaton is "state=1 and fair1_sat", then each fairness constraint is satisfied infinitely often along the fair paths. Such a linearized automaton therefore enforces the fairness requirements while requiring only a single fairness constraint.

We used this linearization technique on each design containing fairness constraints. For each such design, we added an automaton of the above form to the original Verilog description of DesignProp. This automaton had one state for each fairness constraint in the design, plus one state for the fairness constraint of the property (each property contributed exactly one fairness constraint). This new automaton was intersected with DesignProp before minimization; we call this the *OneFair automaton.* The minimization considered only the fairness constraint of this new automaton. As a result, each minimization began with an initial partition containing exactly two equivalence classes (states that satisfied the fairness constraint, and those that did not).

Table 4 summarizes the results of this experiment. The order in which we linearized the fairness constraints had a non-trivial impact on the amount of reduction achieved. For the arbiter, which has three fairness constraints, the number of blocks ranged from 95 to 106 (over six permutations). For bakery-2, which has four fairness constraints, the number of blocks ranged from 1331 to 1417 (over six permutations). For eisenberg-1, which has six fairness constraints, the number of blocks ranged from 3590 to 4130 (over six permutations). We leave the question of how to choose a good ordering to future research.

We can now compare the reductions achieved on each of our three automata: the design (Table 2), the DesignProp (Table 3), and the OneFair (Table 4). With the exception of the bakery example, which achieves slightly better reductions on the original design than on the DesignProp automaton for either property, each example yields more substantial reductions the fewer atomic propositions are considered during minimization.

Conversely, these tables also show an increase in live nodes and memory to minimize the OneFair as opposed to the DesignProp automaton. As in the comparison between Tables 2 and 3, we attribute the increase in the number of live nodes to the additional state variables required by the new automaton. The increase from Table 3 to Table 4, however, is more substantial than the increase noted between Tables 2 and 3. Here, only one experiment (bakery-2) uses fewer live nodes for minimizing OneFair as opposed to DesignProp. The remaining experiments show an average increase of 114% in the number of live nodes.

Based on these contradictory figures, we are unable to say whether the final reduction to a single fairness constraint is worthwhile. Ideally, we would like to find some examples for which the minimization is only tractable under this reduction. Such examples, however, would likely require large numbers of fairness constraints. Our current examples do not meet this requirement. Until we find such evidence, the utility of this extra minimization is questionable.

5 Tuning the Original Algorithm

Attempts to run the previous experiments on even larger designs failed because the runs ran out of memory before finishing. This section discusses an attempt to reduce the algorithm's memory requirements. The Lee-Yannakakis algorithm improves on the original symbolic computation described in Section 3 by processing only those equivalence classes containing reachable states. However, the algorithm still maintains the unreachable classes because they may become reachable as the minimized graph is constructed. We compared the number of reachable classes to the number of classes being maintained at the end of each run of the algorithm; the result appear in Table 5. On average, only 50% of the maintained classes are actually reachable.

This raises a possible approach to improving memory usage. We represent classes as BDDs. In practice, BDDs representing large numbers of states may be much smaller than BDDs representing fewer states. We could therefore try maintaining only a single class containing all those states not yet assigned to a reachable class.[2] Such a merger would create a coarser partition than the Lee-Yannakakis algorithm with respect to the unreachable classes. However, it might require fewer BDD nodes to represent those classes.

Unfortunately, this approach is likely to be computationally expensive. Recall that an unreachable class becomes reachable if it is found to contain a reachable state. In the original algorithm, the previously unreachable class is converted to a reachable block and placed on the stack for processing. The algorithm requires all states in this newly reachable class to agree on the values of the atomic propositions of interest. Maintaining only a single unreachable class violates this requirement. A more complicated procedure would therefore be required to extract a new reachable class that satisfied at least this requirement.

Fortunately, this requirement is sufficient as well as necessary. We could therefore maintain one unreachable class for each combination of atomic propositions. If the number of combinations used is smaller than the number of unreachable classes, we might still save on BDD nodes without incurring a high computational overhead. Our experiments indicate that the number of combinations is indeed substantially smaller than the number of unreachable classes for almost all runs.

Figure 2 shows a modified version of the Lee-Yannakakis algorithm. For each subset of atomic propositions satisfied by some state, the new version maintains a *holding set* of states not currently assigned to any reachable equivalence class. The algorithm views the holding set as a single (large) equivalence class. When a state in a holding set is found to be reachable, the holding set is converted into a reachable block as in the original algorithm. The main change to the pseudocode lies in the "split" section (Figure 1). If the new equivalence class B'' is unreachable at the end of the **foreach** loop, no new block is created and the elements of B'' are returned to the holding set.

Comparing the live node and memory statistics obtained from the original algorithm and our new version yields some interesting results, as shown in Table 6. For the smaller examples, the two algorithms have similar live node and memory figures. For the eisenberg and larger examples, the new algorithm performs substantially worse than the original on the OneFair automata in terms of memory usage (44% worse on average). In contrast, the new algorithm performs substantially better on the DesignProp automata in terms of live nodes (12% better on average, as high as 25% on several examples). The results are mixed for the design minimizations.

[2] Note that this single class does not necessarily contain all of the unreachable states. Reachable classes often contain unreachable states.

	Reachable Classes	Initial Classes	Maintained Classes	% Maintained Reachable
counter	8	4	8	100%
counter-DP	16	2	17	94%
tlc-1	13	27	54	24%
tlc-1-DP	13	6	23	57%
tlc-1-OF	1	2	6	17%
tlc-2	20	18	54	37%
tlc-2-DP	30	6	41	73%
tlc-2-OF	10	2	22	45%
k-elev	132	32	195	68%
k-elev-DP	266	18	311	86%
k-elev-OF	128	2	168	76%
eisenberg-1	1214	1024	4376	28%
eisenberg-1-DP	2074	72	4100	51%
eisenberg-1-OF	3590	2	6509	55%
eisenberg-2	1076	48	2207	49%
eisenberg-2-DP	2713	72	4832	56%
eisenberg-2-OF	2096	2	3887	54%
bakery-(1,2)	2546	48	5485	46%
bakery-1-DP	7258	72	15202	48%
bakery-1-OF	5297	2	8488	62%
bakery-2-DP	5302	72	10533	50%
bakery-2-OF	1417	2	3529	40%
arbiter	64	32	246	26%
arbiter-DP	158	16	450	35%
arbiter-OF	97	2	297	33%
treearbiter4-1	288	4	644	45%
treearbiter4-1-DP	1	2	26	4%
treearbiter4-2	1344	24	2279	59%
treearbiter4-2-DP	3320	32	6586	50%
treearbiter4-2-OF	2365	2	4940	48%
dcnew-1-DP	1	2	2	50%
dcnew-2-DP	242	2	261	93%
coherence-DP	1	2	7	14%

Table 5: Unreachable classes statistics. The DP and OF suffixes denote the DesignProp and OneFair automata, respectively.

```
begin
    stack := {⟨B, i⟩}, where i is the initial state and B is the block containing i;
    queue := ∅ ;
    partition contains only the initial block ⟨B, i⟩ ;
    divide remaining states into one holding set per combination of atomic propositions ;

search:
    while stack ≠ ∅ do begin
        ⟨B, p⟩ := pop(stack) ;
        D := R(B) ;
        blocks := {⟨B', p'⟩ ∈ partition | B' ∩ R(p) ≠ ∅} ∪ {⟨B_i, false⟩|B_i is a holding set and B_i ∩ R(p) ≠ ∅} ;
        foreach block ⟨C, q⟩ in blocks do begin
            if B contains a state with no successor in C then enqueue ⟨B, p⟩ ;
            if q is not defined then select p_c in C ∩ R(p) and set q := p_c ;
            add edge ⟨B, p⟩ → ⟨C, q⟩ ;
            D := D − C ;
        end ;
        if D ≠ ∅ then enqueue ⟨B, p⟩ ;
    end

split:
    while queue ≠ ∅ do begin
        ⟨B, p⟩ := delete (queue) ;
        B' := {q ∈ B : blocks(R(q)) = blocks(R(p))} ;
        B'' := B − B' ;
        B := B' ;
        foreach edge ⟨C, q⟩ → ⟨B, p⟩ in the minimized system do begin
            if some elements of C reach B and some do not then enqueue ⟨C, q⟩ ;
            if some elements of C reach B'' and some do not then enqueue ⟨C, q⟩ ;
            if no successor or q is in B then delete edge ⟨C, q⟩ → ⟨B, p⟩ ;
            if some successor of q is in B'' then
               if there is no block for B'' in the partition then
                  select p''_b in R(q) ∩ B'' ;
                  add ⟨B'', p''_b⟩ to the partition and push it onto the stack ;
            add edge ⟨C, q⟩ → ⟨B'', p''_b⟩ ;
        end
        if there is no block for B'' in the partition then
           add the elements of B'' to the appropriate holding set ;
        if stack ≠ ∅ then goto search ;
    end
```

Figure 2: A refinement on the Lee-Yannakakis combined bisimulation and reachability algorithm [29]. In this algorithm, the partition only contains reachable blocks.

	Max Live Nodes - LY	Max Live Nodes - MLY	Memory (MB) - LY	Memory (MB) - MLY
counter	42	46	2.7	2.7
counter-DP	139	141	2.77	2.77
tlc-1	392	392	2.87	2.88
tlc-1-DP	472	472	2.91	2.91
tlc-1-OF	554	517	2.98	2.98
tlc-2	439	446	2.87	2.88
tlc-2-DP	649	662	2.95	2.95
tlc-2-OF	717	729	3.03	3.03
k-elev	2436	2475	3.06	3.07
k-elev-DP	5035	4947	3.26	3.26
k-elev-OF	12349	11756	3.96	3.98
eisenberg-1	28810	24904	4.59	4.64
eisenberg-1-DP	50447	38289	5.39	5.75
eisenberg-1-OF	113354	121479	11	18
eisenberg-2	28859	24784	4.57	4.68
eisenberg-2-DP	53580	47686	5.72	7.25
eisenberg-2-OF	81224	84095	7.1	10
bakery-(1,2)	68543	50878	5.6	5.42
bakery-1-DP	118871	88355	8.04	8.04
bakery-1-OF	146466	164509	9.26	18
bakery-2-DP	103102	75642	7.4	7.69
bakery-2-OF	78743	88627	6.19	10
arbiter	8477	6820	3.33	3.32
arbiter-DP	21518	22216	3.86	4.01
arbiter-OF	55937	70808	5.55	18
treearbiter4-1	24149	21575	4.7	4.96
treearbiter4-1-DP	19938	14705	6.54	5.22
treearbiter4-2	58805	53648	5.82	6.42
treearbiter4-2-DP	162212	147619	12	17
treearbiter4-2-OF	461598		38	
dcnew-1-DP	2286	2286	4.03	4.03
dcnew-2-DP	15579	16552	4.5	4.6
coherence-DP	18099	17775	7.73	7.73

Table 6: Merged algorithm statistics. In the headings, "LY" indicates the Lee-Yannakakis version and "MLY" indicates our new "merged" class version. MLY figures are not provided for treearbiter4-2-OF because the computation ran out of memory before completing on our machine (128 megabytes).

	Design MC Mem	Design MC Time	Min Design MC Time	Design LY Time	DesProp LY Time	DesProp LE Time	OneFair LY Time	OneFair LE Time
counter	2.6	0.1 (0.1)	0	0.1	0.3	0		
tlc-1	2.76	0.2 (0.2)	0	0.2	0.4	0	0.3	0
tlc-2	2.76	0.2 (0.2)	0	0.4	0.7	0.1	0.6	0
k-elev	2.92	0.5 (0.5)	0.2	4.3	14.9	0.2	39.6	0.2
eisenberg-1*	3.87	3.1 (2.2)	1.4	127.8	287	2.3	1508.9	13.1
eisenberg-2	3.85	6.7 (5.9)	5.9	138.6	438.4	14.7	667.9	36.6
bakery-1*	3.64	8.4 (4.9)	7.8	332.3	1729.3	15.6	1462.8	55.9
bakery-2	3.62	8.4 (4.4)	7.0	315.2	971.5	3.7	455.4	3.6
arbiter	2.98	0.6 (0.5)	0.2	10.7	56	0.5	256.6	0.4
treearbiter4-1	3.7	2.1 (1.1)	0.2	72.3	67.9	0		
treearbiter4-2	4.03	2.1 (1.7)	1.5	466.4	2578.2	7.2	31084.2	14.3
dcnew-1	4.12	1.8 (1.6)	0.2	60.4	1.4	0		
dcnew-2*	4.12	1.8 (1.9)	0.6	59.5	91.7	2.2		
coherence	6.24	45.5 (8.8)			28.2	5.5		

Table 7: Model checking statistics from VIS. The "Design MC Mem/Time" columns indicate the resources used to model check the property in VIS using a monolithic transition relation. The numbers in parentheses in the time column are for partitioned transition relations; there were no significant changes in the memory requirements under the partitioned transition relation. "Min Design MC Time" shows the time taken by model checking after minimizing the design with respect to the atomic propositions in the respective property; this also uses a monolithic transition relation. DesProp abbreviates DesignProp. Memory figures for performing the minimizations appear in Tables 2 through 4. The language emptiness checks did not significantly increase memory usage over that required for minimization. Maximum live node statistics were unavailable for the model checking runs.

6 A Comparison to Model Checking

We evaluate the utility of our minimizations by comparing model checking on minimized systems to model checking on the original designs. We used the VIS CTL model checker to test properties on the original designs (minimized and unminimized) and the VIS language emptiness checker to test for fair paths on the minimized DesignProp and OneFair systems. Our implementation of the Lee-Yannakakis algorithm produces monolithic (rather than partitioned) transition relations. Generating partitioned minimized transition relations is an interesting research challenge. In order to make the comparisons meaningful, we also performed the VIS CTL model checking runs using monolithic transition relations. We were able to run VIS on our minimized designs by replacing the transition relation from the original design with the minimized transition relation in VIS's internal data structures. This works because our minimization does not change how states are represented. The minimized transition relation is merely restricted to relate only the representatives of the reachable equivalence classes.

Table 7 provides four sets of figures. The first two columns give the memory and time needed to model check the CTL properties against the original designs (Table 3 indicates the type of each property). The third column shows the time required for model checking after the design has been minimized with respect to the atomic propositions in the property and in the fairness constraints. The next two columns give the minimization time (under the original Lee-Yannakakis algorithm) and language emptiness test time for the corresponding DesignProp automaton. The last two columns give similar figures for the OneFair automaton. These last two columns are empty for any design that did not have fairness constraints (and thus had no OneFair automaton).

Looking only at the model checking and language emptiness times, the results are mixed. For eisenberg-2, model checking in the original design is actually faster than checking language emptiness in the minimized DesignProp automaton. The eisenberg design has 1611 reachable states (Table 2), while the corresponding

DesignProp automaton has 2713 reachable classes (Table 3). This increase in reachable states explains the increased model checking time. The savings in the remaining cases vary from little to substantial, with the best savings on the coherence example. The original coherence design has 94738 reachable states; after minimization, it has a single reachable block (Table 3). The savings in this case are therefore not surprising.

Performing the language emptiness test on the OneFair automata as compared to the DesignProp automata offers no benefits. In several cases, such as both eisenberg tests and bakery-1, the results are decidely worse for the OneFair automaton. In all three cases, the number of reachable states in the original design is smaller than the number of reachable classes in the corresponding OneFair automaton (Tables 2 and 4).

Accounting for minimization time, performing model checking on the original design is faster than the combination of minimization and language emptiness checking on all but the dcnew1 and coherence examples. Both of these examples have large numbers of reachable states (186876 and 94738, respectively) and reduce to a single reachable block. The total cost of using the OneFair automaton is significantly worse than that for the DesignProp automaton for all but the bakery-2 example.

7 Related Work

Several researchers have explored symbolic minimization techniques. None of these efforts, however, computed reachability simultaneously; they are therefore difficult to use for on-line minimization. Bouali and de Simone provided a symbolic bisimulation algorithm over partitioned transition relations [7]. A variant of this algorithm is implemented in the XEVE verification environment, which also supports VIS's input format [6]. Aziz *et al.* take a compositional approach to bisimulation minimization in a symbolic setting [2]. This approach also performs minimization relative to a given property. However, it represents equivalence relations (rather than classes). Kick uses a symbolic technique to reduce models relative to properties [26]. This technique is not guaranteed to produce transition systems as coarse as those obtained through bisimulation; furthermore, it has not been tested on realistic examples.

Bouajjani *et al.* minimize state graphs from programs using a combination of bisimulation and reachability [5]. Lee and Yannakakis show that the Bouajjani *et al.* algorithm is less efficient than theirs in theory due to the order in which they split classes [29]. In practice, Alur *et al.* found the Bouajjani *et al.* algorithm faster for checking language emptiness of timed automata [1]. Dams proposes a minimization technique for strong preservation of CTL* properties [14]. His technique resembles bisimulation, but produces a potentially coarser relation that preserves only a given set of properties. Rather than split classes based on transitions, he splits a class when its members disagree on subproperties relevant to verifying the given properties. While his approach is fully automatic, it requires many intermediate model-checking analyses, and thus appears computationally prohibitive. Dams, Gerth, and Grumberg present an earlier version of this work for universal CTL formulas [15]. Neither version has been implemented.

Dams, Gerth and Grumberg suggest using abstract interpretations to automatically generate abstractions from program-level descriptions of transition systems [16]. While they aim for strong preservation of formulas, they do not consider minimization relative to particular properties. Furthermore, their generation technique does not necessarily produce the coarsest possible abstraction. This work extends earlier work by Clarke, Grumberg, and Long that considers only weak preservation of universal CTL ($\forall$CTL) properties [10]. Other abstraction frameworks, such as that supported in COSPAN, require the user to manually construct the abstract models [28]. Proposed approaches to symmetry reductions also require various degrees of manual intervention [9, 21, 25]. Furthermore, only Gyuris and Sistla's framework supports fairness constraints.

8 Conclusion and Future Work

We have explored bisimulation minimization as a state-space reduction technique in an automata-theoretic verification framework. Bisimulation is theoretically attractive because it requires no user-intervention and produces the coarsest abstract model that strongly preserves all μ-calculus properties. Practically, however, it has been considered unsuitable for use in symbolic model checking because the BDDs needed to compute the bisimulation relation grow too large.

This paper shows that bisimulation can be viable in a symbolic framework. Using an algorithm due to Lee and Yannakakis, which combines reachability analysis with bisimulation on symbolically-represented transition systems, we have minimized designs far larger than those amenable to conventional bisimulation computations. Much of our success results from being in the automata-theoretic framework. By intersecting the negated property with the design before minimization, we only need to consider the atomic propositions corresponding to fairness constraints during minimization. Reducing the number of atomic propositions used in minimization increases the amount of reduction achieved and the size of design amenable to algorithmic minimization.

As with many proposed techniques to advance the state of model checking [22], our approach works well on some designs and not as well on others. In particular, it appears useful for verifying safety properties of designs with no fairness constraints. Such designs should yield only one reachable equivalence class if the safety property holds. This small number of reachable classes keeps the memory requirements down and reduces the time of convergence for the algorithm. Our approach fits in with a general strategy of providing several model-checking algorithms within a single framework, allowing a user to choose algorithms with a good chance of succeeding based on the nature of the design.

The time and memory requirements of our current approach motivate much of our future work. First, as Table 7 shows, minimization time can be substantial. Alur *et al.* experienced similar time requirements for the Lee and Yannakakis algorithm [1]. Minimizing relative to the DesignProp automata prohibits reuse of results because each property yields a separate machine to be minimized. We therefore plan to experiment with more reusable reductions. For example, if a design has several properties relying on the same few atomic propositions, the design minimization could be performed relative to only those propositions. This reduces the number of atomic propositions used in the minimization as we do here, but yields a more reusable minimized transition system.

Second, the memory requirements for performing our reductions are still significant for larger designs. In particular, the costs of minimization usually outweigh those for performing model checking on the original design. BDD size seems to be the bottleneck to our handling larger examples. Based on analyses not discussed here, the problem appears to lie more in intermediate explosion during reachability computations than in the sizes of the BDDs needed to represent the equivalence classes themselves. We therefore intend to explore alternate symbolic representations in conjunction with the Lee-Yannakakis algorithm.

We have done some preliminary experiments with other forms of decision diagrams, such as multi-terminal BDDs (MTBDDs, ADDs) [3] and zero-suppressed BDDs (ZDDs) [32]. ZDDs offer little to no advantage for this application, but ADDs appear promising. ADDs can represent the equivalence relation as a single structure by relating each state to a constant identifying its equivalence class. While this structure uses more than the number n of state variables, in practice it will not use the $2n$ variables needed to represent the bisimulation relation as a single BDD. Preliminary experiments indicate that the number of ADD nodes in the single structure is often substantially smaller than the sum total of the nodes used to represent the classes separately. Furthermore, a monolithic representation allows us to explore purely symbolic versions of the Lee-Yannakakis algorithm. We have already begun investigations in this area.

Another obvious extension to this work involves integrating the minimization with on-the-fly model checking, as mentioned earlier. On-the-fly model checking is often employed in an automata-theoretic framework [12, 30]. Such integration might improve the utility of this minimization technique with respect to failed properties. We also plan to explore coarser abstractions for the automata-theoretic framework. Bisimulation preserves details about the lengths of paths that are irrelevant for checking whether a fair path exists. Coarser abstractions, such as those based on two-sided simulation [15, 24] or stuttering equivalence [18], may be even better suited to algorithmic state-space reduction in the automata-theoretic framework. Finally, we plan to explore applying our method to abstract part of the state space. For example, it could be used for data path abstraction by collapsing the space of data values to a small set of relevant values [34].

References

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

[3] R. I. Bahar et al. Algebraic decision diagrams and their applications. In *Proc. of the International Conference on Computer-Aided Design*, pages 188–191, 1993.

[4] I. Beer, 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.

[5] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

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

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

[8] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[9] E.M. Clarke, 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.

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

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

[12] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[13] The CUDD package. Available from `http://vlsi.colorado.edu/~fabio/`.

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

[15] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proc. 5th Int.l Conference on Computer-Aided Verification*, pages 479–490, 1993.

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

[17] K. Fisler and M.Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Proc. Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 1998.

[18] J.F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proceedings of the International Conference on Automata, Languages, and Programming*, pages 626–638, 1990.

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

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

[21] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In *Proc. of International Conference on Computer-Aided Verification (CAV)*, pages 232–243, 1997.

[22] R. Hardin, R.P. Kurshan, S. Shukla, and M.Y. Vardi. A new heuristic for bad cycle detection using BDDs. In *Proc. of the International Conference on Computer-Aided Verification*, 1997.

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

[24] T.A. Henzinger, O. Kupferman, and S. Rajamani. Fair simulation. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 273–287, Warsaw, July 1997. Springer-Verlag.

[25] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–76, August 1996.

[26] A. Kick. Formula dependent model reduction through elimination of invisible transitions for checking fragments of CTL. Technical Report 1995-27, Universität Karlsruhe, 1995.

[27] O. Kupferman 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.

[28] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton Univ. Press, 1994.

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

[30] David E. Long. *Model-checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie-Mellon University, 1993.

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

[32] S. Minato. *Binary decision diagrams and applications for VLSI CAD.* Kluwer Academic Publishers, 1996.

[33] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[34] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming*, pages 184–192, St. Petersburgh, January 1986.