Is There a Best Symbolic Cycle-Detection Algorithm?

Kathi Fisler[1,4], Ranan Fraer[2], Gila Kamhi[2], Moshe Y. Vardi[1*], and Zijiang Yang[1,3]

1: Department of Computer Science, Rice University, Houston, TX, USA
2: Intel Development Center, Haifa, Israel
3: CCRL, NEC, Princeton, NJ, USA
4: Worcester Polytechnic Institute, Worcester, MA, USA

Abstract. Fair-cycle detection, a core problem in model checking, is solvable in linear time in the size of the design model using an explicit-state representation. Existing cycle-detection algorithms for symbolic model checking are quadratic or $n \log n$ time in the worst case and often inefficient in practice. Which default symbolic cycle-detection algorithm to implement in model checkers remains an open question. We compare several such algorithms based on the numbers of external and internal iterations and the numbers of image operations that they perform on both randomly-generated and real examples. Unlike recent work by Ravi, Bloem, and Somenzi, we conclude that model checkers need to implement at least two generic cycle-detection algorithms: the traditional Emerson-Lei algorithm and one that evolved from our study, originally due to Hojati *et al.* We demonstrate that these two algorithms are complementary, as the latter algorithm is provably incomparable to Emerson-Lei's and often dominates it in practice.

1 Introduction

Model checking, whether for LTL, CTL, or ω-automata, has linear time complexity in the size of the design model. This well-known result follows from two facts: first, that most model checking techniques reduce to the problem of locating cycles through a given set of nodes in a graph [3, 18]; second, that cycle detection is solvable in linear time using a depth-first search that identifies strongly-connected components (cf, [4]). This depth-first strategy provides a suitable approach to cycle detection in explicit-state model checking, and has been implemented in several tools [7, 11].

Depth-first approaches to cycle detection are not suitable for BDD-based symbolic model checking because BDDs represent sets of states while depth-first search examines individual states. Efficient BDD-based model checking requires efficient breadth-first, set-based cycle-detection algorithms. Most modern symbolic model checkers employ some variant of Emerson and Lei's symbolic cycle-detection algorithm [5]. CTL model checkers use the Emerson-Lei algorithm

[*] Work partially supported by NSF Grant CCR-9988322 and a grant from the Intel corporation.

(henceforth EL) to process formulas of the form $\mathsf{EG}\ \varphi$, which specify infinite paths on which every state satisfies φ. Linear-time model checkers compose the design model with an automaton representing the negation of the property, then check for cycles in the product automaton using the CTL formula $\mathsf{EG}\ \mathbf{true}$. Unfortunately, EL's time complexity is not linear in the size of the design model: the algorithm contains a doubly-nested fixpoint operator, and hence requires time quadratic in the design size in the worst case. The algorithm is also often slow in practice. EL is a so-called *SCC-hull* algorithm [16]. SCC-hull algorithms compute the set of states that contains all fair cycles. In contrast, SCC-enumeration algorithms enumerate all the strongly connected components of the state graph. While SCC-enumeration algorithms have a better worst-case complexity than SCC-hull algorithms [1], their performance in practice seems to be inferior to that of SCC-hull algorithms [16]. This paper focuses on SCC-hull algorithms.

Researchers have proposed several alternatives to EL [8, 10, 14]. Ravi, Bloem, and Somenzi have presented both a classification scheme for such algorithms and an experimental comparison of several algorithms with EL [16]. They concluded that no algorithm consistently outperforms EL for cycle detection, and, consequently, there is no reason to "dethrone" EL as the default cycle-detection algorithm. Their comparison, however, is based primarily on running times, and secondarily on numbers of image operations. This approach has two significant drawbacks: it provides no useful feedback on *why* the algorithms behave as observed, and it suggests no techniques for predicting when one algorithm might outperform another. Furthermore, their comparison considers some algorithms that are based on post-image operations and some that are based on pre-image operations (as is EL), making it rather difficult to draw firm conclusions.

This paper demonstrates a methodology that both addresses these concerns and identifies a symbolic cycle-detection algorithm that provides a viable alternative to EL. Ravi *et al.* present bounds on the number of image operations performed by various cycle-detection algorithms. We argue that to understand the performance of SCC-hull algorithms one needs to measure both the number of image computations as well as the number of external iterations (defined in Section 2). Our methodology focuses on the number of external iterations performed as a basis for comparing and refining symbolic cycle-detection algorithms. In aiming to balance the numbers of external and internal iterations performed, we have identified an algorithm that, as we argue, should join EL as a generic cycle-detection algorithm. We demonstrate that this algorithm is incomparable to EL, dominating it in many cases. Our conclusion is that, as in many other aspects of model checking, there is no "best" cycle-detection algorithm and model checkers need to implement at least both EL and our algorithm.

Section 2 describes our analyses of three existing symbolic cycle-detection algorithms and shows how the competitive algorithm evolved from these analyses. Section 3 presents experimental results on randomly generated and real examples for both the special case of terminal and weak systems and more general examples. Section 4 compares the competitive algorithm to a specialized cycle-detection algorithm for terminal and weak systems. Section 5 concludes.

2 Symbolic Cycle-Detection Algorithms

Cycle-detection algorithms in the context of model checking search for "bad" cycles in a directed graph representing a transition system modeling a design undergoing verification. Two parameters specify which cycles are considered bad: the invariant and the fair sets. The invariant specifies a condition, such as a propositional formula, that must be true of every state on a bad cycle. The fair sets specify sets of states that every bad cycle must pass through. We write $\mathsf{EG}_{\mathsf{fair}}\varphi$ to indicate a search for cycles satisfying invariant φ and passing through fair sets fair. We will omit the fair annotation when all states are considered fair.

Cycle detection in BDD-based model checking is challenging because the BDDs co-mingle information about different paths through a design. Symbolic cycle-detection algorithms maintain a set of states that may lead to bad cycles; this set is conservative, in that it contains all states that do lead to bad cycles. We call this the *approximation set*. The algorithms repeatedly refine the approximation set by locating and removing states that cannot lead to a bad cycle; we call this the *pruning* step. If a state lies on a bad cycle, then it must have a successor and a predecessor on that same cycle (and thus also in the approximation set). Cycle-detection algorithms use this information in different ways.

Formally, these algorithms search for cycles in nondeterministic transition systems. A transition system is a tuple $\langle Q, R, Q_0, \mathcal{F}\rangle$, where Q is a set of states, $Q_0 \subseteq Q$ is the initial state set, $R \subseteq Q \times Q$ is the transition relation, and $\mathcal{F} \subseteq Q$ is the set of fair states. A transition system is *weak* iff (1) there exists a partition of Q into sets $Q_1, \ldots, Q_n$ such that each Q_i is either contained in $\mathcal{F}$ or is disjoint from it, and (2) the Q_i's are partially ordered so that there is no transition from Q_i to Q_j unless $Q_i \leq Q_j$. If the Q_i's contained in $\mathcal{F}$ are the maximal elements of the partial order, a weak system is called *terminal*. This definition of weak and terminal transition systems is due to Bloem, Ravi, and Somenzi [2], as refined from Kupferman and Vardi [15]. In model checking, designs commonly have several fair sets, and bad cycles must pass through each fair set. Such designs are outside the scope of weak systems, whose definition is only meaningful for one fair set.[1]

EL appears in Figure 1 (left).[2] At each iteration through the **while** loop, EL computes the set of states that can reach every fair set via a non-trivial path contained in the approximation set, b. We call these iterations *external*; the reachability computations (the EU formula) form the *internal* iterations. EL does most of its work in the internal iterations: each external iteration performs only one preimage computation per fair set outside of the internal iterations.

Hardin *et al.* attempted to reduce the number of external iterations that EL performs as a means of achieving an improved algorithm [8]. Their algorithm, called Catch-Them-Young (henceforth CTY), aggressively prunes the set

[1] LTL-to-automaton translation algorithms may yield multiple fair sets when one would suffice, rendering an otherwise weak system non-weak. Thus, minimizing the number of fair sets is an important optimization.

[2] Figure 1 shows VIS' implementation of EL; in SMV, the final image computation ($b := b \wedge \mathsf{EX}\ d$) is outside the **for** loop.

```
$b$ := invariant ;
while $b$ changes do
    for each fair set $\mathcal{F}_i$ do
        $d := \mathsf{E}[b\ \mathsf{U}\ (\mathcal{F}_i \wedge b)]$ ;
        $b := b \wedge \mathsf{EX}\ d$ ;
```

```
$b$ := invariant ;
while $b$ changes do
    for each fair set $\mathcal{F}_i$ do
        $\mathcal{F}_i := \mathcal{F}_i \wedge b$ ;
        $b := \mathsf{E}[true\ \mathsf{U}\ \mathcal{F}_i] \wedge \mathsf{E}[true\ \mathsf{S}\ \mathcal{F}_i]$
    while $b$ changes do
        $b := b \wedge \mathsf{EX}\ b \wedge \mathsf{EY}\ b$ ;
$res := \mathsf{EF}\ b$;
```

```
$b$ := invariant ;
while $b$ changes do
    for each fair set $\mathcal{F}_i$ do
        $\mathcal{F}_i := \mathcal{F}_i \wedge b$ ;
        $b := \mathsf{E}[b\ \mathsf{U}\ (b \wedge \mathsf{EX}\ \mathcal{F}_i)]$ ;
    while $b$ changes do
        $b := b \wedge \mathsf{EX}\ b$ ;
```

Fig. 1. The EL (left), CTY (middle), and OWCTY (right) cycle-detection algorithms. In CTY, $\mathsf{EP}\ \mathcal{F}_i$ denotes all states that can reach $\mathcal{F}_i$ and $\mathsf{EY}\ b$ denotes the successors of b. A variant of CTY, CTY+, replaces "true" with b in the EU and ES computations. Each algorithm initializes the approximation set to states satisfying the invariant.

of states potentially lying on bad cycles during the internal iterations (a closely related algorithm was proposed in [10]). This can reduce the number of external iterations by removing states during an external iteration that a later external iteration would otherwise handle in EL.[3] The original CTY algorithm does cycle detection only; it does not compute EG as EL does. For consistency, Figure 1 (middle) provides a version of CTY that can be used to compute EG; this entails one difference from the original algorithm: the extra EF computation in the last step of the algorithm.

The external iterations in CTY perform two steps: first, compute the set of states that are both reachable from and can reach every fair set (the internal iterations); second, repeatedly prune the approximation set until it is closed under both successors and predecessors. In contrast, EL prunes the approximation set only once and removes only states which have no successor in the approximation set; EL does not iterate the pruning step within one external iteration. CTY can eliminate states from the approximation set earlier than can EL, hence the name "Catch-Them-Young". Like EL, CTY has quadratic time complexity with respect to the size of the design. Hardin *et al.*'s experimental results, conducted over a large set of randomly-generated designs, were mixed; CTY tended to outperform EL when there was no bad cycle, but performed worse than EL in the presence of cycles [8]. CTY's aggressive pruning strategy succeeded in reducing the number of external iterations, but nevertheless incurred a noticeable performance penalty.

In order to understand *why* CTY fails to outperform EL, we must examine each algorithm's actual computations. This paper studies patterns of image computations and external iterations, as the former are the most expensive operations in a BDD-based setting and the latter greatly impact the performance of cycle detection algorithms. Section 3 presents numeric data from this analysis. In summary, while CTY performs significantly fewer external iterations than EL, it does not reduce the number of image computations. In essence, EL does too little work outside the internal iterations whereas CTY does too much work overall. Engineering a better balance between the iterations might yield an algorithm that consistently outperforms both EL and CTY. One key difference between EL

[3] Though EL may eliminate states in earlier iterations than CTY.

and CTY is that EL prunes based only on successors, whereas CTY considers both successors and predecessors. An intermediate approach could perform CTY's repeated pruning, but using only pre-image computations, as in EL [19]. This could greatly reduce the number of image computations of CTY, though perhaps at the expense of some additional external iterations. The resulting algorithm, called One-Way-Catch-Them-Young (henceforth OWCTY), appears in Figure 1 (right).[4] OWCTY is essentially the pre-image version of Hojati *et al.*'s EL2 algorithm (sans an initial reachability computation) [10]; its pruning strategy is similar in spirit to that of Kesten *et al.*'s algorithm for cycle detection in the presence of strong fairness [14] (which uses forward instead of backward image operations).

How do OWCTY and EL compare? Hojati *et al.*'s experiments on a small set of small examples discussed only running time and were inconclusive for these two algorithms. Ravi *et al.*'s experiments compared EL and the forward-operator version of EL2/OWCTY; this is not too meaningful, since the issue of forward vs. backward reachability [9] is orthogonal to the balance between external and internal iterations (indeed, the upper bounds obtained in [16] for EL and forward-EL2 are incomparable). OWCTY's worst-case running time has only a linear overhead (see below) over the $\mathcal{O}(|\mathcal{F}|dh)$ worst-case upper bound that Ravi *et al.* identified for EL [16] (where $|\mathcal{F}|$ is the number of fairness constraints, d is the diameter of the state graph, and h is the length of the longest reachable path in the SCC quotient graph). A worst-case analysis as done in [16] provides, however, only a very coarse comparison between the two algorithms. First, the overhead of OWCTY over EL is not very significant. Second, the worst-case instances for EL may be different than those for OWCTY, which means that the comparison of worst-case running times does not tell us how the two algorithms compare on a given input instance. A more meaningful analysis would compare how the two algorithms perform on concrete instances. Analysis at this level shows that the two algorithms are incomparable. Figure 2 illustrates the differences between the EL and OWCTY pruning strategies; OWCTY outperforms EL on the first transition system, while EL outperforms OWCTY on the second.

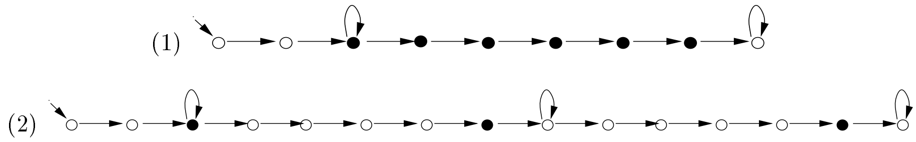

Fig. 2. Two transition systems that illustrate the differences between EL and OWCTY. Black circles denote fair states. All states satisfy the invariant.

Consider the first transition system. Both algorithms eliminate the rightmost state in the first iteration and capture the remaining states in the approximation set. During the first iteration, OWCTY eliminates all but the leftmost fair state;

[4] A variant of OWCTY performs pruning inside the **for** loop; in practice, neither version consistently outperforms the other.

EL eliminates only the rightmost fair state. EL requires an additional iteration to eliminate each of the four middle fair states. Each iteration involves a reachability computation that OWCTY does not perform. If the chain of fair states in the first system contained n fair states, OWCTY would perform $\mathcal{O}(n)$ image computations while EL would perform $\mathcal{O}(n^2)$ image computations. Thus, EL has a quadratic overhead relative to OWCTY on such systems.

Now consider the second transition system. In the first iteration, both algorithms eliminate the rightmost state and retain the remaining states in the approximation set. During the first iteration, EL throws away the rightmost fair state. The reachability computation in the second external iteration begins at the middle fair state; thus, EL eliminates the non-fair states between the right two fair states without traversing them explicitly again. OWCTY, in contrast, uses an additional image computation to eliminate each of those non-fair states. The second system currently contains two copies of a chain of states consisting of four non-fair states, followed by a fair state, followed by a non-fair state with a self loop. If the system had k consecutive copies of this chain, each with m states in the initial non-fair chain, EL would perform $\mathcal{O}(k^2 m)$ image computations as compared to OWCTY's $\mathcal{O}(k^2 m + km) = \mathcal{O}(k^2 m)$ image computations. That is, the overhead of OWCTY relative to EL is only linear.

In general, the two algorithms are incomparable with respect to their numbers of image computations. As OWCTY *provably* performs no more external iterations than EL, OWCTY's overhead (if it exists at all) is caused by the last line of the algorithm, which prunes the approximation set. Thus, OWCTY's overhead is at most linear relative to EL, while, as we saw, EL can have a quadratic overhead relative to OWCTY.

To gain a better picture on the comparative performance of EL, CTY, and OWCTY, the experimental analyses in Section 3 gather data on the numbers of external iterations across several randomly generated and real examples; to complement the Ravi *et al.* study [16], we also include running time, memory usage, and BDD size statistics. Our analyses show that OWCTY requires almost the same number of external iterations as CTY with far fewer image computations; in practice, OWCTY almost always matches or improves on EL's performance.

3 Comparative Analysis of the Algorithms

3.1 Experiments on Random Systems

Our first set of experiments compares the algorithms on random systems. We generate random systems by generating random directed graphs. We would like to obtain directed graphs with non-uniform out-degree and linear density (*i.e.*, a linear number of edges in the number of nodes); linear density prevents cycle detection from becoming trivial due to an excess or paucity of edges. The following model of random graphs, due to Karp [13], satisfies these criteria:

Definition 1 *For each positive integer n and each p with $0 < p < 1$, the sample space consists of all labeled digraphs $D_{n,p}$ with n vertices and edge probability p.*

Given a graph G with vertices V and edges E, the *order* of G is $|V|$ and the *density* of G is $|E|/|V|$. We will use n and d to represent a graph's order and density, respectively. We wish to generate graphs in the space $D_{n,d/n}$. Generating the graphs directly based on this model becomes time consuming as n grows larger: the procedure must decide whether to include each of the possible $n(n-1)$ edges based on the probability d/n. Instead, we fix the number of edges to be the expected number dn, and choose dn distinct edges from the $n(n-1)$ candidates. This approach provides a very good approximation to the given model [19].

Our experiments compare four algorithms: EL, CTY, CTY+, and OWCTY. CTY+ is a variant of CTY that restricts the reachability computations to consider only paths through the approximation set, rather than through the entire state space as in CTY [19]; in other words, CTY+ replaces line 5 of CTY with $b := \mathsf{E}[b \;\mathsf{U}\; \mathcal{F}_i] \wedge \mathsf{E}[b \;\mathsf{S}\; \mathcal{F}_i]$, where S is the past-time operator *since*. We present two sets of results. The first measures the number of external iterations that each algorithm performs, the next measures the number of image computations that each algorithm performs.[5] The experiments use graphs with order 2^{12} and densities varying over 1.2, 1.6, 2.0, and 2.4. This order is large enough to explore the behavior of the algorithms, yet small enough to analyze in a reasonable amount of time. We define a single fair set for each graph, with size varying over $.01n$, $.1n$, $.3n$, $.5n$, $.7n$, and $.9n$ where n is the digraph order. Each experiment fixes either the density or the size of the fair set and varies the other. The figures reported in the rest of this section are averaged over 100 individual experiments.

	$\lvert\mathcal{F}\rvert$			
	$.01n$	$.1n$	$.5n$	$.9n$
CTY	2.18	2.41	2.09	2.00
CTY+	2.18	2.41	2.09	2.00
OWCTY	2.17	2.37	2.07	2.00
EL	2.66	5.36	13.20	20.89

	d			
	1.2	1.6	2.0	2.4
CTY	2.00	2.00	2.00	2.00
CTY+	2.00	2.00	2.00	2.00
OWCTY	2.00	2.00	2.00	2.00
EL	20.89	10.37	7.02	5.09

Table 1. Average number of external iterations on digraphs with order 2^{12}. The left table fixes the density at 1.2 and varies the fair set size. The right table fixes the fair set size at $.9 \times 2^{12}$ and varies the density.

Table 1 shows the number of external iterations on digraphs with order $n = 2^{12}$. One set of experiments fixes the density at 1.2 and varies the fair set size; the other fixes the fair set size at $.9 \times 2^{12}$ and varies the density. The tables indicate that CTY, CTY+ and OWCTY perform far fewer external iterations than EL. Furthermore, OWCTY performs essentially the same number of external iterations as CTY; thus pruning based on predecessors as well as successors, as CTY does, does not significantly reduce the number of external iterations over a pruning strategy based only on successors. We therefore expect OWCTY to consume considerably fewer resources than CTY in practice. EL requires significantly

[5] We refer to post- and pre-image computations collectively as image computations.

more external iterations as the fair set grows larger, and significantly fewer external iterations as the density increases. In contrast, CTY, CTY+, and OWCTY perform fairly consistent numbers of external iterations in both cases.

The data in Table 1 do not indicate that CTY and OWCTY are more efficient than EL because the former algorithms may do more work in the internal iterations. The number of image computations offers a more precise efficiency comparison. Image computations are the most computationally expensive operations in each of the cycle-detection algorithms. The cost of these operations depends on the density and order of the underlying graphs [19]. Since we analyze the four algorithms over the same randomly generated graphs, the cost of individual image computations is comparable across the algorithms. The number of image computations is therefore a fair parameter for comparing the algorithms.

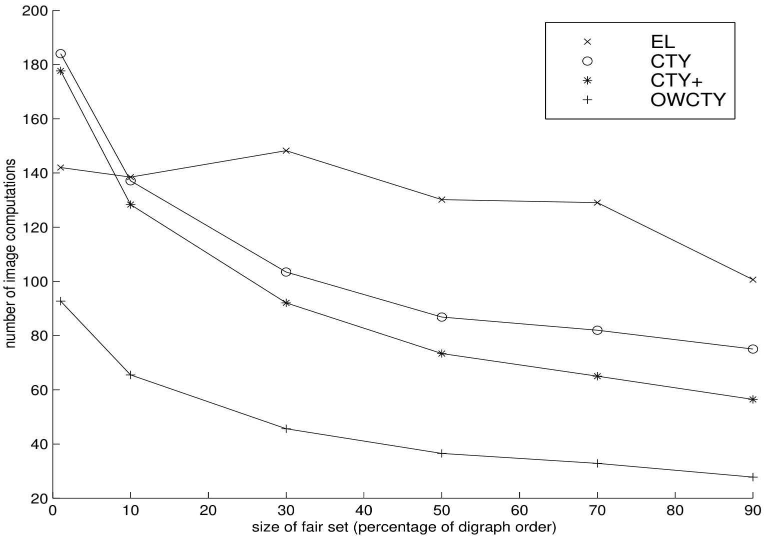

Fig. 3. Number of image computations for EL, CTY, CTY+ and OWCTY.

Figure 3 shows the number of image computations performed over graphs with order $n = 2^{12}$, density $d = 1.2$, and fair set size ranging over $.01n$, $.1n$, $.3n$, $.5n$, $.7n$, and $.9n$. For CTY, CTY+ and OWCTY the number of image computations decreases as the fair set gets larger. CTY performs more image computations than CTY+ because CTY+ restricts reachability computations to the approximation set, which allows the computation to converge faster. OWCTY performs fewer image computations than either CTY or CTY+ because it does not perform forwards reachability. Separate data (not shown) show that the backwards reachability computations in OWCTY and CTY perform almost the same numbers of image computations; furthermore, the pruning step in OWCTY performs roughly half as many image computations as that in CTY+[19]. Thus, eliminating the forward image computations makes OWCTY less computationally expensive without adversely affecting the number of external iterations required.

Separate experiments (not shown) show that the number of image computations decreases sharply as the density increases [19]. In the case of EL, the number of image computations drops because the algorithm performs fewer external iterations as density increases, as discussed previously. For the remaining three algorithms, our experimental data shows that the size of the approximation set after each iteration becomes larger as the density increases. The approximation set determines the base set for subsequent reachability computations. The larger the base set, the faster reachability computations converge [19]. Therefore, fewer image computations are needed when the digraph density increases. Although each pruning step removes fewer vertices, the final approximation set is also larger, so the algorithms perform fewer image computations as density increases. Plots for running time statistics are similar to those for image computations. In particular, both OWCTY and CTY consistently outperform EL. This contradicts the mixed results in other CTY versus EL experiments [8, 16].

3.2 Experiments on Real Systems

Our real design examples come from the VIS distribution and from Fabio Somenzi. They include an ethernet protocol with varying numbers of collisions before failure, a tree-structured arbiter with 8 nodes, a gcd circuit, a floating point multiplier, and two mutual exclusion protocols (bakery and eisenberg). These examples are written in Verilog and evaluated using the VIS model checker [17]. We implemented OWCTY within the VIS framework by replacing the original (EL) algorithm for evaluating EG formulas with OWCTY in a copy of VIS. We ran the experiments using VIS version 1.3 (with version 1.2 of the vl2mv compiler), on an Intel 686 machine with 1GB of memory running RedHat Linux version 2.2.12-20; our VIS installation uses the CUDD BDD package.

Table 2 summarizes experiments with LTL model checking of terminal and weak systems. For each LTL experiment, we evaluated $\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$ on the product of the original design and a manually-constructed automaton for the negation of the property. Table 3 covers examples with multiple fair sets in the context of CTL model checking. Table 4 covers LTL model checking under multiple fairness constraints. In each table, stars on experiment names denote that the models contained cycles or that the property failed. The EX/EY and EU/ES figures count the number of image and reachability computations performed, respectively.[6]

The tables show that OWCTY generally matches or outperforms EL, while CTY and CTY+ are clearly not competitive. In many cases, OWCTY outperforms EL dramatically; in contrast, we have not yet found an example on which EL significantly outperforms OWCTY. The benefits of OWCTY are particularly evident on the ethernet and gcd examples in Table 2. As expected, OWCTY uses fewer external iterations than EL; however, OWCTY sometimes performs more image computations than EL.

[6] The EU/ES counts do not include trivial computations of the form $[\varphi\ \mathsf{U}\ \varphi]$.

Experiment	Procedure	Ext. Iter.	EX or EX/EY	Time (sec)	Mem (MB)	peak live BDD nodes
ethernet 1	EL	51	2179	356.6	13.6	339932
	CTY	2	42/43	187.9	14.6	398280
	CTY+	2	41/42	184.8	14.6	398280
$\mathsf{G}(p \rightarrow \mathsf{F}q)$	OWCTY	3	57	5.5	11.7	175118
ethernet 2	EL	107	6506	10656.1	14.4	367135
	CTY	2	67/68	1893.6	33.6	1365367
	CTY+	2	66/67	1887.6	33.6	1365755
$\mathsf{G}(p \rightarrow \mathsf{F}q)$	OWCTY	3	113	59.6	14.1	404723
ethernet 3	EL	171	11914	4371.3	13.7	279823
	CTY	2	95/96	1962.2	35	1456597
	CTY+	2	94/95	1938.0	35	1456597
$\mathsf{G}(p \rightarrow \mathsf{F}q)$	OWCTY	3	177	24.6	13.8	290593
ethernet 4	EL	-	-	(30H)	-	-
	CTY	2	130/131	5859.7	53.6	2320201
	CTY+	2	130/2	5895	53.6	2320201
$\mathsf{G}(p \rightarrow \mathsf{F}q)$	OWCTY	3	245	491.4	14.1	368225
treearb 8*	EL	8	75	6.2	13.6	234021
	CTY	-	-	(20M)	(23)	-
	CTY+	-	-	(20M)	(23)	-
$\mathsf{G}(p \rightarrow \mathsf{F}q)$	OWCTY	2	24	4.2	12.7	206640
gcd	EL	-	-	(37H)	-	-
	CTY	2	15/3	1384.2	59.3	2298351
	CTY+	2	14/2	1383.0	59.3	2298351
$\mathsf{G}(p \rightarrow \mathsf{XF}q)$	OWCTY	2	24	2497.5	130.9	6285856
fpmult	EL	2	18	18345.8	363	17667058
	CTY	2	26/3	33089.7	369	17619441
	CTY+	2	18/2	21994.7	368	17619441
$\mathsf{G}(p \rightarrow \mathsf{XXX}q)$	OWCTY	2	17	22457.2	369	17422253

Table 2. LTL model checking on weak and terminal systems. Parenthesized times indicate terminated computations; M indicates minutes instead of seconds.

Experiment	Procedure	Num Fair	Ext. Iter.	EX / EU or EX/EY/EU/ES	Time (sec)	Mem (MB)	peak live BDD nodes
bakery1*	EL	6	18	554 / 91	1.3	6.2	34447
	CTY	6	11	1371/650/67/66	10	13.3	176492
	CTY+	6	12	344/299/51/50	7.0	13	182755
$\mathsf{AG}(p \rightarrow \mathsf{AF}q)$	OWCTY	6	18	516 / 75	1.6	6.1	36962
bakery2	EL	6	18	490 / 92	1.3	6.0	29524
	CTY	6	11	1239/614/67/66	9.4	13.3	176492
	CTY+	6	11	282/246/47/46	6.0	12.7	180657
$\mathsf{AG}(p \rightarrow \mathsf{AF}q)$	OWCTY	6	18	444 / 72	1.4	5.8	28849
treearb8*	EL	8	15	382 / 106	14.8	13.6	328115
	CTY	8	-	-	(194M)	(112)	-
	CTY+	8	-	-	(170M)	(123)	-
$\mathsf{AG}(p \rightarrow \mathsf{AF}q)$	OWCTY	8	13	416 / 104	13.1	13.4	309449
eisenberg2	EL	6	27	669 / 124	1.6	5.5	17352
	CTY	6	23	2159/2031/139/138	7.8	11.2	180311
	CTY+	6	16	252/506/56/55	3.8	8.6	148353
$\mathsf{AG}(p \rightarrow \mathsf{AF}q)$	OWCTY	6	27	631 / 102	1.4	5.4	18504
elevator*	EL	8	12	849/97	498.2	13.8	275914
	CTY	8	-	-	(104M)	(38)	-
	CTY+	8	-	-	(104M)	(43)	-
$\mathsf{AG}(p \rightarrow \mathsf{AF}q)$	OWCTY	8	12	861/79	536.8	13.6	275914

Table 3. CTL model checking on systems with multiple fairness constraints.

Experiment	Procedure	Num Fair	Ext. Iter.	EX / EU or EX/EY/EU/ES	Time (sec)	Mem (MB)	peak live BDD nodes
treearb8*	EL	9	15	1021 / 135	1397.8	13.8	239731
	CTY	9	-	-	(186M)	(44)	-
	CTY+	9	-	-	(207M)	(157)	-
$\mathsf{G}(p \rightarrow \mathsf{F}q)$	OWCTY	9	14	1000 / 126	911.6	13.9	369062
eisenberg2	EL	7	24	1332 / 161	5.7	7.2	47704
	CTY	7	24	5114/5486/169/168	60.3	13.7	240028
	CTY+	7	15	229/399/53/52	4.7	8.8	147763
$\mathsf{G}(p \rightarrow \mathsf{F}q)$	OWCTY	7	24	1197 / 109	5.3	7.1	59802
elevator3*	EL	3	2	7 / 1	1164.7	87.5	4062730
	CTY	-	-	-	(60M)	(270)	-
	CTY+	-	-	-	(60M)	(270)	-
$\mathsf{G}p$	OWCTY	3	2	13 / 1	1167.3	87.5	4062730
elevator4*	EL	1	2	3 / 1	16192.4	282	13308496
	CTY	1	-	-	(365M)	(278)	-
	CTY+	1	-	-	(367M)	(278)	-
$\mathsf{G}p$	OWCTY	1	2	5 / 1	16388.0	282	13308496

Table 4. LTL model checking on systems with multiple fairness constraints.

Exp.	Proc.	Num Fair	Ext. Iter.	EX	Time (sec)
A*	EL	2	6	203	65.49
	OWCTY	2	2	77	32.58
D*	EL	6	2	147	16.26
	OWCTY	6	2	149	16.33
E*	EL	4	3	125	6.89
	OWCTY	4	2	87	6.39
F*	EL	2	10	50	870.0
	OWCTY	2	2	27	897.7
H1*	EL	2	8	40	633.8
	OWCTY	2	2	23	495.7
H3*	EL	2	8	40	550.5
	OWCTY	2	2	23	592.7
I*	EL	2	2	40	1004.5
	OWCTY	2	2	23	692.9
J1*	EL	2	8	40	521.9
	OWCTY	2	2	23	426.6
J2*	EL	2	8	40	447.9
	OWCTY	2	2	23	347.7
K*	EL	2	7	25	220.3
	OWCTY	2	2	20	165.3
L*	EL	2	6	24	129.4
	OWCTY	2	2	19	129.4
M1*	EL	2	7	35	81.5
	OWCTY	2	2	21	53.9

Table 5. Results from Intel on checking $\mathsf{EG}_{\mathsf{fair}}$**true** on systems that have (and require) multiple fairness constraints.

Finally, we compared OWCTY and EL on Intel designs using internal Intel tools (Table 5). All the table entries reflect the composition of actual designs with linear-time properties, using multiple fairness constraints. OWCTY performed significantly better than EL in all examples except F and H3, where EL slightly outperformed OWCTY.

4 OWCTY Versus Specialized Algorithms

Our experimental results show that OWCTY generally outperforms EL on terminal and weak systems. Bloem, Ravi, and Somenzi have presented an algorithm that is specialized to verify terminal and weak systems efficiently [2]. Linear-time model checkers detect bad cycles by using the EL algorithm to check EG **true** over the product of the design and the negation of the desired property. Bloem *et al.* observed that for terminal and weak systems, CTL formulas capture the search for bad cycles. Specifically, the formulas $\mathsf{EF\ fair}$ and $\mathsf{EF\ EG\ fair}$ are true of terminal and weak systems, respectively, when they contain infinite fair cycles. Accordingly, their algorithm (henceforth BRS) checks one of the formulas $\mathsf{EF\ fair}$, $\mathsf{EF\ EG\ fair}$, or $\mathsf{EG}_{\mathsf{fair}}$**true** based on the structure of the input system. This structure follows from the structure of the property being tested: if a property corresponds to a weak (resp. terminal) system, the product of that property and a design model is also a weak (resp. terminal) system. Bloem *et al.* showed that BRS significantly outperforms EL in practice on terminal and weak systems.

Table 6 compares OWCTY to BRS.[7] For the examples from Table 2, we checked both $\mathsf{EG}_{\mathsf{fair}}$**true** and the appropriate formula from BRS using OWCTY. The statis-

[7] The gcd and fpmult examples are the same as Bloem *et al.* used in their paper [2]. Our resource usage on these examples differs widely from theirs due to differences between our two versions of the compiler from Verilog to BLIF, the VIS input language.

Experiment	Procedure	EX	Time (sec)	Mem (MB)	peak BDD nodes
ethernet 1	$\neg$EF EG fair	53	4.2	11.2	151306
$\mathsf{G}(p \to \mathsf{F}q)$	$\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$(OWCTY)	57	5.5	11.7	175118
ethernet 2	$\neg$EF EG fair	109	24.4	13.7	381839
$\mathsf{G}(p \to \mathsf{F}q)$	$\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$(OWCTY)	113	59.6	14.1	404723
ethernet 3	$\neg$EF EG fair	173	13.3	13.6	287787
$\mathsf{G}(p \to \mathsf{F}q)$	$\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$(OWCTY)	177	24.6	13.8	290593
ethernet 4	$\neg$EF EG fair	241	145.6	14.0	373531
$\mathsf{G}(p \to \mathsf{F}q)$	$\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$(OWCTY)	245	491.4	14.1	368225
treearb 8*	$\neg$EF EG fair	22	4.1	12.6	200529
$\mathsf{G}(p \to \mathsf{F}q)$	$\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$(OWCTY)	24	4.2	12.7	206640
gcd	$\neg$EF EG fair	20	3351.6	193	8204281
$\mathsf{G}(p \to \mathsf{XF}q)$	$\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$(OWCTY)	24	2497.5	130.9	6285856
fpmult	$\neg$EF fair	8	5565.5	329	16109729
$\mathsf{G}(p \to \mathsf{XXX}q)$	$\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$(OWCTY)	17	22457.2	369	17422253

Table 6. Comparison between the OWCTY and BRS algorithms.

tics on $\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$ are reproduced from Table 2. The specialized approach outperforms OWCTY on most of these examples (except the gcd example). This is due to the difference between checking $\mathsf{EG}_{\mathbf{true}}\mathsf{fair}$ (BRS) and $\mathsf{EG}_{\mathsf{fair}}\mathbf{true}$ (OWCTY). The former restricts the search for a bad cycle to the fair states; the latter looks for a cycle that intersects the fair states. As a result, both EL and OWCTY can have non-fair states in their approximation sets, while BRS' approximation set contains only fair states. This restriction usually allows BRS to converge faster.

This comparison demonstrates how exploiting structural information about systems can lead to more efficient verification algorithms. Note, however, that BRS is not a generic cycle-detection algorithm. Furthermore, we must also consider the cost of determining whether a system is weak or terminal, which is not included in our paper or in Bloem *et al.*'s. In theory, this operation can be done symbolically in $\mathcal{O}(n \log n)$ time [1], but experimental results are not yet available. For the simple properties considered by Bloem *et al.* and here, this overhead is insignificant; for more complicated properties (such as those including complex environmental assumptions) it could be rather substantial. OWCTY, which is a generic algorithm, performs well in practice without the overhead of specialized analyses as required in BRS.

5 Conclusions

Symbolic model checking remains a heuristic process, as metrics do not yet exist to predict BDD behavior under differing algorithms. As a result, comparative analyses of algorithms are extremely useful in helping tool developers choose which algorithms to implement. In the name of good science, these analyses need to be reproducible and portable to the greatest extent possible. Such analyses provide not only firm data, but a foundation for future algorithm development.

This paper compares three symbolic cycle-detection algorithms (and a variant on one of them) based on the number of iterations they take through their outermost fixpoint operator, as well as the number of image operations they perform. Each algorithm employs a slightly different strategy for pruning the set of states potentially lying on cycles. Our analysis shows that the original Emerson-Lei (EL) algorithm [5] performs too little work outside of its internal iterations, while Hardin *et al.*'s Catch-Them-Young (CTY) algorithm [8] performs too much. In contrast, Hojati's EL2 algorithm [10], which we view as a one-way version of CTY (OWCTY) does seem to balance the work inside and outside the internal iterations. On random examples and on terminal and weak systems, OWCTY dominates EL, while on general systems, OWCTY is competitive with EL, dominating it significantly in many cases. We have also shown that the two algorithms are incomparable with respect to the number of image computations they perform: EL can have a quadratic overhead over OWCTY, while OWCTY can have a linear overhead over EL. These results support our conclusion that model checkers need to contain both EL and OWCTY.

In the course of this project, we have identified two desired features for verification tools. First, we want tools to implement multiple algorithms for common problems such as cycle-detection. Both our analysis and the recent one by Ravi *et al.* [16] indicate that no algorithm consistently outperforms the others; indeed, verification tasks may be tractable with one algorithm and intractable with another. Tools providing multiple algorithms afford human verifiers opportunities to experiment and find algorithms that work on their applications. A similar conclusion in the context of semi-exhaustive reachability analysis was reached in [6]. Second, we want tools to provide visualizations of computational patterns during model checking. Intel's Palette [12] does some of this; we wish we had such a tool to augment VIS and other publicly-available tools. Testbeds supporting multiple algorithms and better data collection would provide strong support for more disciplined approaches to algorithm comparisons in verification.

Acknowledgements

We thank Kavita Ravi, Fabio Somenzi, and Roderick Bloem for their very helpful comments on this paper, and the Rice PLT group for access to their large-memory server.

References

1. Bloem, R., H. N. Gabow and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Intl. Conf. on Formal Methods in Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

2. Bloem, R., K. Ravi and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *Intl. Conf. on Computer-Aided Verification*, Lecture Notes in Computer Science, pages 222–235. Springer-Verlag, 1999.
3. Clarke, E. M., E. A. Emerson and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.
4. Courcoubetis, C., 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.
5. Emerson, E. A. and C. L. Lei. Efficient model checking in fragments of the propositional model mu-calculus. *Proceedings of LICS 1986*, pages 267–278, 1986.
6. Fraer, R., G. Kamhi, L. Fix and M. Y. Vardi. Evaluating semi-exhausting verification techniques for bug hunting. In *Proceedings of the 1st Intl. Workshop on Symbolic Model Checking*. Electronic Notes in Theoretical Computer Science, 1999.
7. Hardin, R. H., Z. Har'El and R. P. Kurshan. COSPAN. In *Intl. Conf. on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 423–427. Springer-Verlag, 1996.
8. Hardin, R. H., R. P. Kurshan, S. K. Shukla and M. Y. Vardi. A new heuristic for bad cycle detection using BDDs. In *Proc. Conf. on Computer-Aided verification (CAV'97)*, pages 268–278. Springer-Verlag. LNCS 1254, 1997.
9. Henzinger, T., O. Kupferman and S. Qadeer. From *pre*historic to *post*modern symbolic model checking. In Hu, A. and M. Vardi, editors, *Intl. Conf. on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 195–206. Springer-Verlag, 1998.
10. Hojati, R., H. Touati, R. Kurshan and R. Brayton. Efficient ω-regular language containment. In *Intl. Conf. on Computer-Aided Verification*, number 663 in Lecture Notes in Computer Science. Springer-Verlag, 1992.
11. Holzmann, G. and D. Peled. The state of SPIN. In *Intl. Conf. on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 385–389. Springer-Verlag, 1996.
12. Kamhi, G., L. Fix and Z. Binyamini. Symbolic model checking visualization. In *Intl. Conf. on Formal Methods in Computer-Aided Verification*, number 1522 in Lecture Notes in Computer Science, pages 290–303. Springer-Verlag, 1998.
13. Karp, R. M. The transitive closure of a random digraph. *Random Structures and Algorithms*, 1(1), 1990.
14. Kesten, Y., A. Pnueli and L. on Raviv. Algorithmic verification of linear temporal logic specifications. In *Intl. Colloquium on Automata, Languages, and Programming*, number 1443 in Lecture Notes in Computer Science. Springer-Verlag, 1998.
15. Kupferman, O. and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *IEEE Symp on Logic in Computer Science*, 1998.
16. Ravi, K., R. Bloem and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Intl. Conf. on Formal Methods in Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 2000.
17. The VIS Group. VIS: A system for verification and synthesis. In Alur, R. and T. Henzinger, editors, *Intl. Conf. on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1996.
18. Vardi, M. Y. and P. Wolper. An automata-theoretic approach to automatic program verification. In *IEEE Symposium on Logic in Computer Science*, 1986.
19. Yang, Z. Performance analysis of symbolic reachability algorithms in model checking. Master's thesis, Rice University, Department of Computer Science, 1999. Available at `http://www.cs.rice.edu/CS/Verification/`.