V. Conclusion

We have presented a complete planar clock routing system called ACTD, which is implemented by employing heuristic techniques. Our first algorithm, called CLE routing, is implemented to solve the overlapping of clock net routing and prevent the crossing in the clock net. The second algorithm, called POA routing, reconstructs the clock tree using heuristics, to avoid obstacles with the scheme of changing tapping points. We have validated our techniques experimentally using test circuits. Future work includes extending the CLE routing and POA routing approach to include well-balanced clock trees, since our current implementation constructs a planar clock routing in the presence of obstacles with allowing the path length skew. Extending our present results to the minimum skew goal presents an intriguing direction for future work. We expect that partitioning line adjustment based on the weight of load and tapping point adjustment for a given planar clock tree will help to minimize clock skew.

References

[1] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors* San Jose, CA, 1998.

[2] M. Edahiro, "A clustering-based optimization algorithm in zero-skew routings," in *Proc. 30th ACM/IEEE Design Automation Conf.*, 1993, pp. 612–616.

[3] K. Boese and A. Kahng, "Zero-skew clock routing trees with minimum wirelength," in *Proc. IEEE 5th Int. ASIC Conf.*, 1992, pp. 1.1.1–1.1.5.

[4] J. Cong, L. He, C. Koh, and P. Madden, "Performance optimization of VLSI interconnect layout," *INTEGRATION, the VLSI J.*, vol. 21, Aug. 1996.

[5] Q. Zhu and W. Dai, "Planar clock routing for high performance chip and package co-design," *IEEE Trans. VLSI Syst.*, vol. 4, pp. 210–226, June 1996.

[6] A. Kahng and C. Tsao, "Low-cost single-layer clock trees with exact zero Elmore delay skew," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1994, pp. 213–218.

[7] X. Zeng, D. Zhou, and W. Li, "An effective buffer insertion algorithm for high-speed clock network," in *Proc. IEEE Int. Symp. Physical Design*, 1999, pp. 172–175.

[8] W. C. Elmore, "The transient response of damped linear network with particular regard to wideband amplifier," *J. Appl. Phys.*, vol. 19, pp. 55–63, 1948.

Implicit Enumeration of Strongly Connected Components and an Application to Formal Verification

Aiguo Xie and Peter A. Beerel

***Abstract*—This paper first presents a binary decision diagram-based implicit algorithm to compute all maximal strongly connected components (SCCs) of directed graphs. The algorithm iteratively applies reachability analysis and sequentially identifies SCCs. Experimental results suggest that the algorithm dramatically outperforms the only existing implicit method which must compute the transitive closure of the adjacency-matrix of the graphs. This paper then applies this SCC algorithm to solve the bad cycle detection problem encountered in formal verification. Experimental results show that our new bad cycle detection algorithm is typically significantly faster than the state-of-the-art [1], sometimes by more than a factor of ten.**

***Index Terms*—Bad cycle detection, binary decision diagrams, formal verification, strongly connected components, symbolic methods.**

I. Introduction

Decomposing a directed graph (digraph) into its (maximal) strongly connected components (SCCs) is a fundamental graph problem [2] and has many important applications in computer-aided design. Generally speaking, SCC decomposition often divides a digraph problem into subproblems, one for each SCC. The solution to the original problem can be constructed by combining the solutions to the subproblems, sometimes with the aid of the component graph (i.e., the structure of connections among SCCs).

Using an explicit data structure such as an *adjacency-list* or an *adjacency-matrix* [2], the decomposition of a digraph $G(V, E)$ (with V being the set of its nodes and E the set of its edges) can be done in linear time (i.e., $O(|V| + |E|)$) using a depth-first search [2], [3]. However, in many real applications, the size of the digraph can be too large (e.g., with more than 10^{20} nodes) for explicit methods to be practical. One promising alternative is to use an implicit representation of the graph, for example using binary decision diagrams (BDDs) [4]. Such implicit representations are often dramatically smaller than their explicit counterparts and facilitate graph algorithms on very large digraphs.

An implicit BDD-based method to find all SCCs in the reachable state space of a finite state machine (FSM) is described in [5].[1] It first implicitly computes the transitive closure of the state transition relation of the machine, and then computes all SCCs *simultaneously*.[2] We call this method TC-based. In practice, however, despite the advantages of the implicit data structures, computing the transitive closure has been shown to be very computationally expensive in both CPU time and memory.

This paper proposes an alternative implicit approach that is motivated by related algorithms that identify only the *terminal* SCCs, i.e.,

Manuscript received January 28, 2000; revised May 5, 2000. This work is supported in part by the National Science Foundation (NSF) under CAREER Award MIP-9502386 and Award CCR-9812164, and in part by Semiconductor Research Corporation (SRC) under Grant 98-DJ-486. This paper was recommended by Associate Editor E. Cerny.

A. Xie is with Cadence Design Systems, Inc., San Jose, CA. 95134 USA.

P. A. Beerel is with the Electrical Engineering-Systems Department, University of Southern California, Los Angeles, CA 90089-2562 USA (e-mail: pabeerel@usc.edu).

Publisher Item Identifier S 0278-0070(00)09153-3.

[1]The method serves as an intermediate step in finding a subclass of SCCs called terminal SCCs in the paper.

[2]A similar idea is also exploited in [6] to find terminal SCCs of states in a FSM.

those from which no other SCCs can be reached [7]–[9]. These algorithms, identify all terminal SCCs *sequentially* by iteratively applying reachability analysis [7]–[9], rather than extracting the terminal SCCs from the set of all SCCs found using the above TC-based method. Because implicit reachability analysis is much less computationally expensive than implicitly computing the transitive closure and FSMs often have a limited number of terminal SCCs, these algorithms have been shown to be much more efficient than the TC-based method in finding terminal SCCs.

In particular, this paper explores whether such a BDD-based reachability analysis approach can be extended to sequentially find *all* SCCs (not just the terminal SCCs) of a digraph. This paper answers this question in the affirmative by presenting an algorithm that recursively partitions the digraph into subgraphs using reachability analysis and reduces the SCC identification problem to individual subgraphs that are sequentially analyzed. Experiments show that the algorithm is dramatically faster and solves much larger problems than the TC-based method, especially when the graph has a small number of SCCs. In contrast, it should be understood that this sequential approach will not work well for digraphs containing numerous, very small SCCs. In fact, for such graphs, explicit methods based on depth-first search may be more efficient.

This paper also explores the application of our new SCC algorithm to the bad cycle detection problem, encountered in many formal verification problems, stated as follows [10], [1]. Given a set of state sets, called *cycle sets*, determine if there is any reachable cycle of states that is *not* contained in at least one cycle set. Such a cycle is called a *bad cycle* because it reflects a cyclic behavior that is unexpected and/or unwanted. The current implicit state-of-the-art algorithm to solve this problem is by Hardin *et al.* [1]. This method iteratively refines the set of bad states using *over approximations* of SCCs. In contrast, we propose to first use our new algorithm to find all SCCs *exactly* and then verify whether each SCC is contained in at least one cycle set. In particular, there exists a bad cycle if and only if there exists an SCC that is not contained in at least one cycle set. The key intuition behind this fact is that the SCC itself forms a (possibly nonsimple) cycle that is not covered by any cycle set. We present experimental results on VIS benchmark examples that indicate our algorithm is generally much faster than the algorithm in [1].

The remainder of this paper is organized as follows. Section II reviews implicit representation of digraphs using BDDs, and summarizes the TC-based method. Section III details our reachability-analysis (RA)-based method. The performance of the two methods are compared in Section IV. Section V describes its application to the bad cycle detection problem and Section VI concludes the paper.

II. Preliminaries and the TC-Based Method

Let $G(V, E)$ denote a digraph where V is the set of its nodes, and for any two nodes $u, v \in V$, $(u, v) \in E$ iff there is an edge from u to v. A vector $(v_1, v_2, \ldots, v_{n+1})$ of nodes is a *path* of length $n \geq 1$ (from v_1 to v_{n+1}) iff $(v_i, v_{i+1}) \in E$, for all $i = 1, \ldots, n$. Notice that in contrast to some traditional definitions [2], we do not consider a vector of length one to be a path. Node v is *reachable* from u (denoted by $u \rightsquigarrow v$) if there is a path from u to v. A (maximal) *strongly connected component* (SCC) is a (maximal) subset of nodes where every pair of nodes are reachable from each other. Below, when we say SCCs, we refer to the maximal ones. We denote by $\mathcal{A}(G)$ the set of SCCs in G. Notice that based on our definition of a path, a node may not be in any SCC, which we refer to as a non-SCC node. All other nodes are SCC nodes. This definition is convenient since in most applications we are aware of these non-SCC nodes are not of interest. However, note that all our definitions and proposed algorithms can be easily altered to accommodate the more traditional definition.

Let the nodes be labeled with distinct numbers from $\{1, 2, \ldots, |V|\}$. The digraph can then be represented by an adjacency matrix $M_{|V|\times|V|}$ [2] whose element $M(u, v)$ is one if $(u, v) \in E$, and zero otherwise. The digraph may also be represented by a BDD $N(X, Y)$ by encoding the rows and columns of its adjacency matrix M with two sets of BDD variables X and Y. The BDD $N(X(u), Y(v))$ evaluates to one if $M(u, v) = 1$ where $X(u)$ and $Y(v)$ are the vectors encoding the labels of u and v, respectively. Details of BDDs and their common operations can be found in [4]. For convenience, we sometimes use node u to refer to the BDD encoding of its labeling.

The TC-based method first computes the transitive closure of N, denoted by N^*, by iteratively squaring N using standard BDD operations.[3] By definition, $N^*(u, v) = 1$ iff $u \rightsquigarrow v$. The transpose of N^* denoted by N^{*t} has the property that $N^{*t}(u, v) = 1$ iff $v \rightsquigarrow u$. Thus, u and v, $u \neq v$ belong to the same SCC iff $N^*(u, v) \wedge N^{*t}(u, v) = 1$ [4]. Consequently, the union of all SCC nodes (denoted by H) can be computed as $H(X) = \exists_Y N^*(X, Y) \wedge N^{*t}(X, Y)$, where $\exists_Y$ denotes BDD *existential quantification over variable set* Y [4]. Note that existential quantification of a BDD B over a variable y reduces to a disjunction as follows: $\exists_y B(y) = B(0) \vee B(1)$. Moreover, existential quantification of a BDD B over a set of variables Y is the disjunction of the existential quantifications of B over all variables $y \in Y$ [4]. Finally, the SCC containing a particular node v can be computed as $\exists_Y X(v) \wedge N^*(X, Y) \wedge N^{*t}(X, Y)$.

III. The RA-Based Method

Computing the transitive closure of the adjacency-matrix of a digraph as required by the TC-based method is equivalent to computing the reachability set of every node (the set of nodes reachable from the given node). Our method which is based on reachability analysis (the RA-based method below) needs to compute the reachability sets of potentially very few nodes. To better describe our method, we need to introduce a few notations related to reachability sets and several properties of such sets.

A. Forward and Backward Sets

The forward set $F(v)$ of a node $v \in V$ is the set of nodes reachable from v. That is, $F(v) = \{u \in V \mid v \rightsquigarrow u\}$. Similarly, its backward set $B(v)$ is the set of nodes that can reach v. That is, $B(v) = \{u \in V \mid u \rightsquigarrow v\}$. One of the properties of $F(v)$ and $B(v)$ is given in Lemma 3.1, which states that the intersection of the two sets (if not empty) is an SCC. Additionally, v is a non-SCC node if the intersection is empty.

Lemma 3.1 [7], [9], [8], [6]: If v is a node of $G(V, E)$, then $F(v) \cap B(v) \in \mathcal{A}(G)$.

A subset $U \subseteq V$ is *SCC-closed* if no SCC intersects with both U and its complement $V \setminus U$. That is, any SCC must be either completely contained in such a set or they do not overlap.

Lemma 3.2: Both backward and forward sets of any node are SCC-closed.

Proof: Since a digraph and its (edge-) reversed graph have the same set of SCCs, we just need to show that any backward set is SCC-closed. Consider a node u, and assume its backward set $B(u)$ is not SCC-closed. Thus, there is a nonempty SCC, say A such that $A \cap B(u) \neq \emptyset$ and $A \setminus B(u) \neq \emptyset$. Let x and y be any nodes of $A \cap B(u)$ and $A \setminus B(u)$, respectively. Since x and y are nodes of A, y must be a node of $B(x)$. Moreover, since x is a node of $B(u)$, $B(x) \subseteq B(u)$

[3]Note that symbolic iterative squaring does not directly take advantage of the sparse nature of the graph often used in explicit methods.

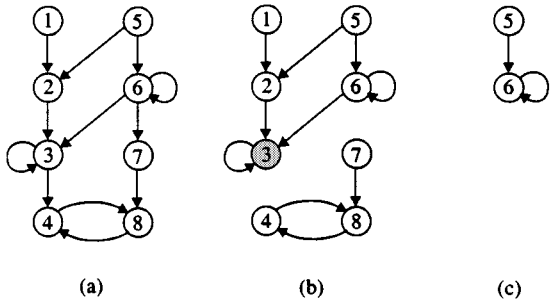

Fig. 1. A digraph and its decomposition.

[7]. Thus, y is a node of $B(u)$, which means $A \backslash B(u) \subseteq B(u)$. The latter can be true only if $A \subseteq B(u)$ or $A = \emptyset$, which contradicts the assumption. □

Theorem 3.1: The difference of the backward sets of any two nodes is SCC-closed.

Proof: Let u and v be any two nodes. Without loss of generality, let us prove the difference $D = B(v) \setminus B(u)$ is SCC-closed. If $D = B(v)$ or $D = \emptyset$, the result is trivial. Now suppose $D \neq B(v)$ and $D \neq \emptyset$, but it is not SCC-closed. Then, there must be an SCC, say A such that $A \cap D \neq \emptyset$ and $A \setminus D \neq \emptyset$. Since $B(v)$ is SCC-closed by Lemma 3.2, we have $A \subseteq B(v)$ and, thus, $A \setminus D \subseteq B(v)$. Consequently, $\emptyset \neq A \setminus D \subseteq B(v) \setminus D \subseteq B(u)$. Therefore, $A \cap B(u) \neq \emptyset$. Besides, $\emptyset \neq A \cap D = A \cap (B(v) \setminus B(u)) \subseteq A \cap (V \setminus B(u)) = A \setminus B(u)$. Hence, $B(u)$ is not SCC-closed. This contradicts Lemma 3.2. □

Remark: By similar arguments, one can show that the difference of any two of $B(u)$, $B(v)$, $F(u)$ and $F(v)$ is SCC-closed, and more generally, that the difference of any two SCC-closed sets is SCC-closed.

Example: In the digraph shown in Fig. 1(a), $B(3) = \{1, 2, 3, 5, 6\}$, $F(3) = \{3, 4, 8\}$, and $B(4) = \{1, 2, 3, 4, 5, 6, 7, 8\}$. From Lemma 3.1, $B(3) \cap F(3) = \{3\}$ is an SCC. From Lemma 3.2, $B(3)$ is SCC-closed. Moreover, the difference $B(4) \setminus B(3) = \{4, 7, 8\}$ is also SCC-closed from Theorem 3.1.

B. The Algorithm

Lemma 3.2 and Theorem 3.1 suggest that a digraph may be first partitioned into backward sets (or differences of backward sets) of some of its nodes, and then the computation of SCCs of the graph can be restricted to each of the partitions. This leads to a divide-and-conquer strategy. For instance, the digraph in Fig. 1(a) can be partitioned into $B(3)$ and $B(4) \setminus B(3)$, which reduces the problem to finding SCCs within the two subsets independently, as illustrated in Fig. 1(b).

Because a digraph G and its transpose G^t (the graph obtained by reversing all the edges of G) has the same set of SCCs [2], the above divide-and-conquer strategy may also be carried out in terms of forward sets. In fact, the partitioning is also possible to be carried out by interleaving the computation of backward sets and forward sets. However, our experience shows that none of these three different partitioning schemes leads to a significant advantage over the others. For this reason, we focus on explaining the algorithm based on the backward-set partitioning scheme. We note that the above divide-and-conquer strategy may also be implemented using an explicit data structure, but would be impractical for large graphs as in the explicit depth-first-search algorithm.

At the top level, the algorithm first picks a node from the set of remaining nodes (the entire set V at the beginning), computes its backward set restricted to the remaining nodes, and then decomposes the backward set into SCCs and a set of non-SCC nodes. This process repeats until the remaining set of nodes is empty.

The procedure is formally described in Fig. 2. In the algorithm, function `random_take`(V') randomly picks a node from the set

```
SCC_DECOMP(N, V)
    V'←V;
    while(V' ≠ 0) {
        v←random_take(V');
        B(v)←backward_set(v, V', N);
        SCC_DECOMP_RECUR(v, B(v), N);
        V'←V' ∧ ¬(v ∨ B(v));
    }
```

Fig. 2. The top-level algorithm.

```
FMD_PRED(W, U, N)
    pred←0;
    front←W;
    bound←U;
    while(front ≠ 0) {
        front←bound ∧ pre(front) ∧ ¬pre(bound);
        pred←pred ∨ front;
        bound←bound ∧ ¬front;
    }
    return pred;
```

Fig. 3. Computing the predecessors with finite maximum distance.

V'. Function `backward_set`(v, V', N) returns the backward set of v *restricted to* set V', which can be implemented efficiently using a fixed-point computation [11]. The remainder of this section explains the core procedure `SCC_DECOMP_RECUR`$(v, B(v), N)$ which recursively decomposes $B(v)$ into SCCs and a set of non-SCC nodes.

To describe the procedure `SCC_DECOMP_RECUR`, we introduce a concept of predecessors with finite maximum distance, also referred to as *FMD predecessors*. A node u is an FMD predecessor of node v if any path from u to v has finite length. That is to say, any path from u to v must not pass any SCC. A necessary condition for this is that u must be a non-SCC node. The set of FMD predecessors of a set of nodes W is denoted $\mathrm{FMD\_PRED}(W)$. As an example, in Fig. 1(b), $\mathrm{FMD\_PRED}(\{3\}) = \{1, 2\}$. Node 5 is not a FMD predecessors of node 3 because it may pass the SCC $\{6\}$ to reach node 3.

Fig. 3 gives an implicit algorithm that iteratively computes the set $\mathrm{FMD\_PRED}(W)$ restricted to a set U given the digraph representation N. The algorithm uses an upper bound of the set of FMD predecessors called *bound* that is initialized to U. In each iteration it computes *front*, the subset of nodes in *bound* that are *immediate* FMD predecessors of the *front* computed in the previous iteration, where *front* is first initialized to W. In other words, in each iteration, *front* is updated to be the subset of nodes in *bound* that can in one step reach (the previously computed) *front* but not *bound*. Moreover, in each iteration, *front* is added to the set of currently identified FMD predecessors *pred* (which is initialized to be empty) and removed from *bound*. The iterations stop when *front* is empty at which time *pred* is returned. Note that in the algorithm *pre*(S) denotes the immediate predecessors of a set of nodes S and is easily implemented using basic BDD operations using the digraph representation N.

Example: Consider the computation of $\mathrm{FMD\_PRED}(\{3\})$ in Fig. 1(b) restricted to the set $U = \{1, 2, 5, 6\}$. Initially, *bound* is initialized to U and *front* is initialized to $\{3\}$. In the first iteration of the while loop, *front* is updated to be the set of node that can immediately reach $\{3\}$ but no node in U, i.e., *front* is set to be $\{2\}$. Notice that node 2 is a FMD predecessor of node 3. Moreover, notice that node 6 is not in this set because although it can immediately reach node 3 it can also reach itself which is in U. After removing *front* from *bound* and adding *front* to the set of FMD predecessors *pred*, a second iteration is initiated. In this iteration, *front* is updated to be $\{1\}$

```
SCC_DECOMP_RECUR(v, B(v), N)
    F(v)←forward_set(v, B(v), N);
    if(F(v) ≠ 0)
        report F(v) an SCC;
    else
        report v non-SCC;
    x←F(v) ∨ v;
    R←B(v) ∧ x̄;
    y←FMD_PRED(x, R, N);
    report y non-SCC;
    R←R ∧ ȳ;
    IP←pre(y ∨ x) ∧ R;
    while(R ≠ 0) {
        v←random_take(IP);
        B(v)←backward_set(v, R, N));
        SCC_DECOMP_RECUR(v, B(v));
        R←R ∧ \overline{v ∨ B(v)};
        IP←IP ∧ \overline{v ∨ B(v)};
    }
```

Fig. 4. Recursive decomposition of a backward set.

because node 1 is the only node that can both immediately reach $\{2\}$ (the previously computed *front*), but reach no other node in *bound*. In particular, node 5 is excluded from *front* because it can reach node 6 in *bound*. After again updating the variables, the third iteration finds no node that can reach $\{1\}$ which terminates the algorithm, yielding the final set of FMD predecessors to be $\{1, 2\}$, as desired.

The recursive procedure SCC_DECOMP_RECUR$(v, B(v), N)$ works as follows. It first computes the forward set $F(v)$ of v restricted to $B(v)$. If $F(v)$ is not empty, then it is an SCC due to Lemma 3.1. Otherwise, node v is a non-SCC node. In either case, $F(v) \vee v$ can be removed from $B(v)$ from further consideration. Next, the FMD predecessors of $F(v) \vee v$ is computed, and are subsequently removed from further consideration. Then, from the remaining set of nodes, the procedure computes the set of immediate predecessors (IP) of these already removed nodes. To decompose the remaining set of nodes, a node u is randomly picked from the IP set and its backward set $B(u)$ is computed. The procedure then calls itself to decompose $B(u)$. After it returns, u and $B(u)$ are removed from the remaining set of nodes and the IP set is updated. If the remaining set of nodes is not empty, another node w is picked up from the updated IP set, and its backward set $B(w)$ is recursively decomposed. This process repeats until the remaining set of nodes is empty. A formal description of the procedure is given in Fig. 4.

Example: In Fig. 1(b), a call to SCC_DECOMP_RECUR$(3, B(3), N)$ first computes $F(3)$ restricted to $B(3)$ by computing forward_set $(3, B(3), N)$, and reports the result $F(3) \cap B(3) = \{3\}$ to be an SCC. Next, it computes FMD_PRED$(\{3\}, \{1, 2, 5, 6\}, N) = \{1, 2\}$, as described above, which is reported to be a set of non-SCC nodes, and the IP set is computed to be $\{5, 6\}$. At this point, the procedure has reduced the problem to decomposing the digraph in Fig. 1(c). Suppose node 6 is picked from the IP set, the procedure calls SCC_DECOMP_RECUR$(6, B(6), N)$ which, in this case, leaves no more nodes to be decomposed when it returns.

C. The Complexity

As shown in Fig. 4, procedure SCC_DECOMP_RECUR performs a constant number of reachability analyzes before it calls itself. Each reachability analysis must have completed in $O(\mathcal{D})$ BDD operations where $\mathcal{D}$ is the diameter of the graph. Since a call to SCC_DECOMP_RECUR removes at least one node from further consideration, SCC_DECOMP_RECUR$(v, B(v), N)$ cannot call itself for more than $|B(v)|$ times. Thus, a trivial upper bound on the complexity of our algorithm is $O(|V| \times \mathcal{D})$ in BDD operations. Notice that this is larger than the $O(V + E)$ complexity of explicit methods using depth-first search. Moreover, a single BDD operation may be dramatically more expensive than a single operation on an explicit data structure. Consequently, in some cases, BDD-based methods can be more expensive than explicit methods. That said, our experimental results suggest that the typical run-time of our algorithm is much better than this worst case complexity analysis suggests.

TABLE I
THE EXPERIMENTAL RESULTS WHERE $|V|$ IS THE NUMBER OF REACHABLE STATES, $|V'|$ DENOTES THE NUMBER OF STATES BELONGING TO SCCs, mo DENOTES MEMORY OUT AND to DENOTES TIME OUT AFTER 1 HOUR OF CPU TIME

	State space			CPU secs	
Example	$\|V\|$	$\|V'\|$	SCCs	TC	RA
abp	484	444	11	5.41	0.40
arbit	5,568	5,568	1	797	0.52
bakery	2,886	2,604	31	197	3.24
coherence	94,748	94,688	2	mo	41.0
dcnew	186,876	139,520	3,469	to	202
eisenberg	1,611	1,609	1	to	0.49
emodel	34,133	376	12	to	96.0
elevator	674,077	92,584	10,435	to	894.3
scheduler	2.4e+6	2.4e+6	1	mo	1.19
minMax8	2.8e+6	2.8e+6	1	mo	0.37
minMax16	4.7e+13	4.7e+13	1	mo	0.84
minMax32	1.3e+28	1.3e+28	1	mo	8.50
8-arbit	>4.0e+7	-	-	to	to
tcp	3.9e+22	-	-	mo	to

IV. EXPERIMENTS

We have implemented both TC-based and RA-based methods with vis-1.3 [12]. In particular, the transitive closure of the adjacency-matrix of the graph is implicitly computed by iteratively squaring the matrix. This section compares their performance using the examples distributed with the vis-1.3 package each of which specifies a FSM. The methods are used to compute the SCCs in the reachable state space of the FSMs. The package is run on a Sun Ultra 10 with 640 Mbytes of memory. Table I gives the experimental results of a representative set of the examples.

Since the TC_method simultaneously computes all SCCs, we report its run time as soon as it computes the transition closure of the adjacency-matrix and ANDs the transitive closure with its transpose. That is, the time to list all the SCCs is not included in the reported run times of the TC_method.

As shown in the Table, the TC-method solves only a few small examples whereas our RA-based method solves most of them. Even for those solved by the TC-based method, the RA-based method is faster by orders of magnitude. These results are reflective of the fact that the BDD-based method for computing the transitive closure [12] is typically much more expensive than the BDD-based methods for reachability analysis. The RA-based method does particularly well in those examples whose state space contain only one SCC. This scenario seems to occur often, potentially because many hardware circuits have a built-in reset signal which implies that the reachable state space is an SCC.

Our RA-based method runs out of time in the last two examples for different reasons. In the *8-arbit* example, the reachability analysis we perform starting from a randomly selected state takes an excessively

long time, despite the fact that analysis from the large set of given initial states is relatively quick. This difference may be due to the fact that our method may be searching longer sequences of states and/or because the intermediate BDD obtained during our search from *individual* random states may be bigger than those obtained during reachability analysis from *all* given initial states. In the last example, *tcp*, the RA-based method times out because the example has too many SCCs that must be reported, highlighting a key limitation of our sequential search strategy. Note also that for examples that have relatively small number of nodes and many SCCs (e.g., the *elevator* example), explicit methods might be more efficient than our RA-based symbolic method. However, for very large examples, explicit methods are simply not applicable.

V. Application to the Bad Cycle Detection Problem

The bad cycle detection problem encountered in many formal verification tasks is stated as follows. Given a set of state sets, called *cycle sets*, determine if there is any reachable cycle of states that is *not* contained in at least one cycle set. Such a cycle is called a *bad cycle* because it reflects a cyclic behavior that is unexpected and/or unwanted. States in any bad cycle are referred to as *bad states* and the set of all bad states is denoted S_B.

The current implicit state-of-the-art algorithm to find S_B is by Hardin *et al.* [1]. They observed that a cycle is bad if and only if it touches the complements of all cycle sets. First, for each complement $\overline{c_i}$ of each cycle set c_i, they compute the the intersection of its backward and forward sets $F(\overline{c_i}) \cap B(\overline{c_i})$. For all i, the ith intersection is a superset of all the SCCs that intersect $\overline{c_i}$ and therefore must contain all bad states, i.e., S_B. They then take the intersection of all these supersets and observe that the result must still be a superset of all bad states. From this intersection, they iteratively remove states that either do not have a predecessor or do not have a successor that is not in this set, since these states cannot be part of a bad cycle. This yields a refined superset of S_B denoted $\hat{S}_B$. They then restrict all complement sets to $\hat{S}_B$ and repeat the entire procedure. The iteration terminates when $\hat{S}_B = \emptyset$, in which case there is no bad cycle, or when $\hat{S}_B$ does not change from one iteration to the next. In the latter case, $\hat{S}_B = S_B$.

In contrast, we propose to first find all SCCs (exactly) and then verify whether each SCC is contained in at least one cycle set. In particular, there exists a bad cycle if and only if there exists an SCC that is not contained in at least one cycle set. The key intuition behind this fact is that if an SCC exists that is not contained in at least one cycle set, then the SCC itself forms at least one (possibly nonsimple) cycle that is not covered by any cycle set.

For comparison, we implemented both algorithms with `vis-1.3` and tested them on a number of the example circuits included in the `vis-1.3` package. Following the experimental setup proposed in [1], we created two versions of the cycle sets. The first cycle set version consists simply of the set for SCCs in the reachable state set. This version creates a passing example for the bad cycle detection problem because all cycles are contained in at least one cycle set. The second cycle set version is created from the first by removing one of the cycle sets which creates a failing example. For each failing example, we created two experimental setups. The first in which all bad cycles were detected, referred to as *fail-all*, and the second when the algorithm is terminated as soon as one bad cycle is detected, referred to as *fail-one*. The fail-one algorithm will terminate no later than the fail-all algorithm and the difference is determined by the order of SCCs explored. For this reason, we randomly chose the removed cycle set over ten different times and averaged the run-times for both algorithms. Moreover, for each example, we report the average number of iterations needed by Hardin *et al.*'s algorithm.

TABLE II
Speedup of Our New Bad Cycle Detection Algorithm over the State of the Art [1]

Example	Ave. # Iter [1]	Pass	Fail-All	Fail-One
abp	4	1.8	2.5	3.0
arbit	1.2	1	1	1
bakery	7	10.2	14	21
coherence	2	0.7	1.5	3.5
dcnew	6.8	2	2.1	3.0
eisenberg	2	4.5	1	6
emodel	6.5	3	5	15.2
minMax32	1.2	1	1	1

Table II gives the speedup of our algorithm over Hardin *et al.*'s. Note that the speedup is generally higher when the (average) number of iterations used by Hardin *et al.*'s algorithm is large. This is to be expected because each iteration of Hardin *et al.*'s algorithm has a similar amount of computation as does our entire algorithm, particularly in systems with small numbers of SCCs. In fact, even for systems with large numbers of SCCs, such as the example *dcnew*, our algorithm still significantly outperforms Hardin *et al.*'s algorithm. We also note that because, like ours, Hardin *et al.*'s algorithm avoids computing the transitive closure of the state space, both algorithms use comparable amounts of memory.

Finally, we point out the existence of a dual of our algorithm that also uses the observation that a reachable cycle is bad if and only if it intersects with the complements of all cycle sets. The idea is to use the new SCC algorithm to first identify all SCCs that intersect with the complement of the first cycle set. Then, verify whether any such SCC intersects with all remaining complement sets. If such an SCC is found, it forms a bad (possibly nonsimple) cycle. Otherwise, no bad cycle exists. Moreover, heuristics can be used to guide the choice of the first cycle set to improve the run-time. We do not present the run-time results of this dual algorithm since its implementation yielded no significant run-time differences from the proposed algorithm.

VI. Conclusion

Computing SCCs of a directed graph is a generic graph problem. For this purpose, we have proposed an implicit algorithm using BDDs. The method iteratively applies reachability analysis and computes SCCs sequentially. It has been used to compute the SCCs in the reachable state space of a set of FSMs. Compared with an existing symbolic method which requires the transitive closure of the adjacency-matrix of the graph, our method is shown to be much faster, requires much less memory, and consequently solves much larger problems.

We applied our algorithm to the *bad-cycle-detection* problem [10], [1]. Our experimental results on a number of VIS benchmarks indicate that our approach is significantly faster than the state-of-the-art described in [1].

Finally, we observe that when there are excessive SCCs, our approach may not complete in a reasonable amount of time. Consequently, as future work, it might be interesting to explore a hybrid approach between our RA-based method and the TC-based method, which may simultaneously find multiple SCCs of relatively small size without computing the transitive closure of the entire graph.

Acknowledgment

The authors would like to thank F. Somenzi of the University of Colorado at Boulder for insightful and motivating discussions regarding

SCC identification. They would also like to acknowledge the anonymous reviewers of earlier versions of this manuscript for providing many useful suggestions for improvement.

REFERENCES

[1] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi, "A new heuristic for bad cycle detection using BDDs," in *Proc. Int. Workshop Computer-Aided Verification*, 1997, pp. 268–278.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[3] R. E. Tarjan, "Depth first search and linear graph algorithms," *ACM J. Computing*, vol. 1, no. 2, 1972.

[4] R. E. Bryant, "Graph-based algorithm for boolean function munipulation," *IEEE Trans. Comput.*, vol. C-35, Aug. 1986.

[5] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Markovian analysis of large FSMs," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1479–1493, Dec. 1996.

[6] G. Hasteer, A. Mathur, and P. Banerjee, "An implicit algorithm for finding steady states and its applications to FSM verifcation," in *Proc. ACM/IEEE Design Automation Conf.*, 1998, pp. 611–614.

[7] A. Xie and P. A. Beerel, "Efficient state classification of finite state Markov chains," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 1334–1338, Dec. 1998.

[8] V. Singhal, "Design Replacements for Sequential Circuits," Ph.D. dissertation, Univ. California, Berkeley, 1996.

[9] S. Qadeer, R. K. Brayton, V. Singhal, and C. Pixley, "Latch redundancy removal without global reset," in *Proc. Int. Conf. Computer Design (ICCD)*, Oct. 1996, pp. 432–439.

[10] E. A. Emerson and C. L. Lei, "Efficient model checking in fragments of the propositional modal mu-calculus," in *Proc. LICS 1986*, 1986, pp. 267–278.

[11] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 401–424, Apr. 1994.

[12] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Kharti, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, "VIS: A system for verfication and synthesis," in *Proc. Int. Conf. Computer-Aided Verification*, 1996, pp. 428–432.