An I/O Efficient Approach for Detecting All Accepting Cycles

Lijun Wu, Kaile Su, Shaowei Cai, Xiaosong Zhang, Chenyi Zhang, and Shupeng Wang

Abstract—Existing algorithms for I/O Linear Temporal Logic (LTL) model checking usually output a single counterexample for a system which violates the property. However, in real-world applications, such as diagnosis and debugging in software and hardware system designs, people often need to have a set of counterexamples or even all counterexamples. For this purpose, we propose an I/O efficient approach for detecting all accepting cycles, called Detecting All Accepting Cycles (DAAC), where the properties to be verified are in LTL. Different from other algorithms for finding all cycles, DAAC first searches for the accepting strongly connected components (ASCCs), and then finds all accepting cycles of every ASCC, which can avoid searching for a great many paths that are impossible to be extended to accepting cycles. In order to further lower DAAC's I/O complexity and improve its performance, we propose an intersection computation technique and a dynamic path management technique, and exploit a minimal perfect hash function (MPHF). We carry out both complexity and experimental comparisons with the state-of-the-art algorithms including Detect Accepting Cycle (DAC), Maximal Accepting Predecessors (MAP) and Iterative-Deepening Depth-First Search (IDDFS). The comparative results show that our approach is better on the whole in terms of I/O complexity and practical performance, despite the fact that it finds all counterexamples.

Index Terms—Model checking, detection of all accepting cycles, state space explosion, accepting strongly connected component, breath-first search

1 Introduction

Model checking has become one of the most attractive and important approaches to verification for software and hardware systems. The automata-theoretic approach, as one of the most important model checking techniques, translates the LTL model checking problem into the detection of reachable accepting cycles in a directed graph [1]. However, with the increasing scale and complexity of software and hardware systems, the directed graph tends to be extremely large and induces the state explosion problem. One of the most effective approaches to this problem is to exploit external memory devices (disks), which is called the external-memory approach, and has been extensively studied [2], [3], [4], [5], [6], [7], [8], [9], [10], [11].

Compared to internal memory, external memory devices (disks) can provide much larger space. Moreover, in the past few year, there have been enormous increases in the capacity of magnetic disks, with little increase in their cost, resulting in dramatic reductions in the cost per byte. Magnetic disks are about two and a half orders of magnitude cheaper than semiconductor memory [12]. These two facts together suggest the idea of using external memory in model checking large-scale systems.

Previous I/O LTL model checking approaches are usually designed to find one counterexample, as one counterexample is sufficient to judge the invalidity of the model [2], [3], [4]. However, in real-world applications, such as diagnosis and debugging in system designs, people often need to have a set of counterexamples or even all counterexamples.

To demonstrate that all (or a set of) counterexamples can be helpful in locating software bugs, let us observe the following example. Suppose a software engineer wrote a program to compute $x^2 + 6x + 9$ where x is an integer from -100 to $+100$, and his codes are as follows:

$scanf$("%d", x);
y = x × x;
y = y + 6x;
y = y − 9;

Apparently, he mistakenly typed $+9$ as -9. When debugging the program, he finds the following three counterexamples:

1) $s_{11}(x = -1, y = 0), s_{12}(x = -1, y = 1), s_{13}(x = -1, y = -5), s_{14}(x = -1, y = -14)$;

- *L. Wu is with the School of Computer Science and Engineering, University of Electronic Science and Technology, Chengdu, China, and the School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia. E-mail: wljuestc@sina.com.*
- *K. Su is with the Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia, and the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China. E-mail: shaowei001@sina.com.*
- *S. Cai is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. E-mail: wljuestc@yahoo.com.*
- *X. Zhang is with the School of Computer Science and Engineering, University of Electronic Science and Technology, Chengdu, China. E-mail: gzzsudocwlj@sina.com.*
- *C. Zhang is with the School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Australia. E-mail: Chenyi001@sina.com.*
- *S. Wang is with the School of Computer Science and Engineering, University of Electronic Science and Technology, Chengdu, China. E-mail: shupengwanguestc@sina.com.*

Manuscript received 28 May 2014; revised 14 Feb. 2015; accepted 1 Mar. 2015. Date of publication 8 Mar. 2015; date of current version 26 Aug. 2015. Recommended for acceptance by A. Roychoudhury.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2411284

2) $s_{21}(x=0,y=0), s_{22}(x=0,y=0), s_{23}(x=0,y=0), s_{24}(x=0,y=-9)$;
3) $s_{31}(x=1,y=0), s_{32}(x=1,y=1), s_{33}(x=1,y=7), s_{34}(x=1,y=-2)$.

Now he wants to locate the wrong code line by these three counterexamples. Clearly, he may not complete the work only by one counterexample. However, when considering all three counterexamples, he may find the difference of the correct function value and the output of the program is a constant 18 in these three counterexamples. Thus it is most likely that the line containing a constant addend is written mistakenly, and he concludes that the fourth line is most likely the wrong code line. In this case, the three counterexamples together can provide more useful information than any one of them does.

Indeed, a lot of work about applications of a set of counterexamples in diagnosis and debugging of software and hardware systems has been carried out [13], [14], [15], [16], [17]. Leue and Befrouei developed an automated method for explaining counterexamples indicating the occurrence of deadlocks in concurrent systems [13]. The method is based on an analysis of a set of counterexamples that can be generated by a model checking tool. The set of counterexamples is referred as the bad dataset. With the aid of the model checking tool, the method also gets a set of execution traces that do not violate the property, which is referred as the good dataset. By examining the differences in the good and bad datasets, it extracts a number of ordered sequences of actions, which point to the root causes of the deadlock occurrence in the model. Fady Copty et al. presented a diagnosis and rectification solution based on a set of counterexamples [14]. The usage flow of the solution consists of running a model checker, dumping all the model checking data needed to compute all the counterexamples of a given length, and then debugging in an interactive environment by loading the pre-dumped model checking data. The efficiency of their approach were evaluated through the case studies on Intel's actual design [14].

Thus it is very important to develop efficient approaches for locating root bugs (or root causes of bugs) by all or sufficiently many counterexamples. Therefore, to find all or sufficiently many accepting cycles and to efficiently perform debugging in software and hardware system designs by these accepting cycles are two closely related and significant problems.

On the other hand, for large models, though it could take several minutes for a model checker to find a counterexample, a user may find it convenient to get all (or a set of) counterexamples, and then fix them.

This paper focuses on investigating an external memory approach for finding all accepting cycles of large-scale systems. We outline the main technical contributions of this paper as follows.

1) To the best of our knowledge, we are the first to propose an I/O efficient approach that detects all accepting cycles of large-scale systems, called detect all accepting cycles (DAAC). DAAC can be very helpful for debugging in software and hardware designs, as the information contained in the found counterexamples can be better used to identify root bugs accurately.
2) This paper provides a general framework for finding all counterexamples of a large-scale system, which is different from previous algorithms [18], [19], [20], [21], [22], [23]. Previous algorithms search for all elementary cycles directly from the directed graph. In our framework (see Section 4), in order to find all accepting cycles, DAAC traverses the accepting state set. For each accepting state, DAAC first searches for the ASCC containing the accepting state, and then finds accepting cycles of the ASCC. When all accepting states are traversed, all accepting cycles of the system are found. The framework has the advantage that it avoids searching for a great many ineffective paths, where an ineffective path is a path that can not be extended to an accepting cycle. This framework has essential contributions to the good performance of our approach, as shown in Section 4.
3) To further lower the I/O complexity and improve the performance of our approach, we propose a technique for computing the intersection of two state sets and a technique for dynamic path management (DPM). The intersection computation technique has linear worst-case I/O complexity and thus can improve the performance of the algorithm that searches for ASCCs. The dynamic path management technique can reduce the memory dithering phenomenon during search, and thus can improve the performance of the algorithm that finds all accepting cycles of an ASCC. The memory dithering in this paper refers to the phenomenon that states are frequently moved into and out of the internal memory, which may significantly increase the number of I/O operations and thus reduce the efficiency of an algorithm (see Section 4.3 for more details).

To demonstrate the effectiveness of our approach, we compare our approach with Detect Accepting Cycle (DAC) [2], Maximal Accepting Predecessors (MAP) [3], [5] and Iterative-Deepening Depth-First Search (IDDFS) [4], in terms of both I/O complexity and practical performance. DAC, MAP and IDDFS are I/O efficient LTL model checking algorithms that achieve state-of-the-art performances and represent the most recent advances. The comparative results show that our approach is better on the whole in term of I/O complexity and practical performance, despite the fact that it finds all counterexamples.

This paper is organized as follows. We discuss related work in the next section, and introduce some preliminary notions in Section 3. Section 4 proposes an I/O efficient approach for detection of all accepting cycles, followed by correctness proof in Section 5. In Sections 6 and 7, we compare our approach with DAC, MAP and IDDFS, in terms of both I/O complexity and performance. Section 8 discusses the parameter of dynamic path management and on-the-fly approaches that directly search for all (or a set of) accepting cycles from the graph, and explores two variants of DAAC. We conclude this paper in Section 9.

2 Related Work

In this section, we introduce existing algorithms for I/O LTL Model Checking and those for finding all elementary cycles.

2.1 I/O LTL Model Checking

The first I/O-efficient algorithm for the LTL model checking was proposed by Edelkamp and Jabbar [6]. The algorithm reduces the accepting cycle detection problem to the reachability problem [24], which is designed for symbolic exploration with BDDs. Afterwards, many I/O efficient algorithms for solving the LTL model checking problem were proposed [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. To the best of our knowledge, among this kind of algorithms, DAC [2], MAP [3], [5] and IDDFS [4] achieve state-of-the-art performance and represent the most recent advances.

The algorithm DAC [2] adapts an existing non-DFS-based accepting cycle detection algorithm One Way Catch them Young (OWCTY) [25] to the I/O efficient setting. The algorithm first inserts all reachable vertices into an approximation set. After that, it repeatedly reduces the approximation set until a fixpoint is reached. In detail, vertices violating the condition are gradually removed from the approximation set using two procedures. One procedure removes those vertices from the approximation set that lies outside any cycle. The other removes vertices lying on non-accepting cycles. Finally, if the approximation set is empty, there is no accepting cycle in the graph, otherwise the presence of an accepting cycle is ensured. The algorithm is especially useful for verification of large systems with valid properties. Nevertheless, it needs to create the whole state space.

Since DAC does not work in an on-the-fly way, Barnat et al., the same authors of DAC, further proposed an on-the-fly algorithm: MAP algorithm [3], [5], which is a revisiting resistant algorithm for I/O efficient LTL model checking. Revisiting resistant graph algorithms are those that can tolerate reexploration of edges without yielding incorrect results. The main idea behind the MAP algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own accepting predecessor (an accepting predecessor of a state t is an accepting state from which t is reachable). Instead of expensive computing and storing of all accepting predecessors for each (accepting) vertex, the algorithm computes and stores a single representative accepting predecessor for each vertex, namely the maximal one in a suitable ordering of vertices. Experiments showed the algorithm outperformed previous I/O efficient algorithms on invalid LTL properties.

IDDFS [4] is a 5-bit semi-external LTL model checking algorithm proposed by Edelkamp et al. Semi-external graph algorithms are algorithms in which the vertices but not the edges fit in memory [26]. IDDFS uses heuristic EPH to construct a minimal perfect hash function (MPHF) from the vertex set stored on disk, which allows compressing V to $5|V|$ bits, and only needs to store the $5|V|$ bits but not V into internal memory, where V is the vertex set of a graph, and $|V|$ is the size of V. Thus the algorithm can handle spaces that are orders of magnitudes larger than internal memory. However, IDDFS still has a limitation on the size of the graph because it needs 5 bits of internal memory for every vertex. This algorithm works on-the-fly by applying iterative-deepening strategy.

The main difference between our approach and the I/O LTL model checking approaches above is as follows: our approach is designed to detect all accepting cycles, while the latter are designed to detect one accepting cycle, and can require that every state may not be repeatedly visited. As our approach needs to find all accepting cycles, states may be on different accepting cycles, and so may be repeatedly visited, which will increase significantly the searching time. This requires our approach for higher efficiency.

2.2 Algorithms for Finding All Elementary Cycles

The algorithm elementary circuit (EC) proposed by Tiernan [18] is one of the earliest algorithms for finding all elementary cycles in a directed graph. The algorithm finds all elementary cycles by a backtracking procedure which explores elementary paths of the graph and check if they are elementary cycles. The vertices of the graph are numbered from 1 to n, where n is the number of vertices in the graph. All elementary paths begin from different vertices. The first path starts from vertex 1. A path is extended from its end, one arc at a time. In the extension process, the index of the extended vertex must be larger than that of the last vertex of the path. If the index of the extended vertex is equal to that of the first vertex of the path, then the path is an elementary circuit (i.e. cycle). The algorithm enumerates each elementary cycle exactly once, because each cycle contains a unique vertex with smallest index. The time complexity of this algorithm is exponential in the size of the graph.

Tarjan [19] improved EC algorithm and gave a bound of $O(n \cdot e(c+1)))$ for the running time of the algorithm on an entire graph in the worst case, where n is the number of vertex; e is the number of edges; and c is the number of elementary circuits in the graph. The idea of this algorithm is as follows. For each vertex s, the algorithm generates an elementary path p which starts at s and contains no vertex whose index is smaller than that of s. The elementary path p is stored in a point stack. A vertex v is marked if it is on the elementary path p or if every path leading from v to s intersects p at a point other than s. A vertex v becomes unmarked if the algorithm finds a path from v to s which does not intersect p other than at s. Once a vertex v has been used in a path, it can only be used in a new path when it has been deleted from the point stack and when it becomes unmarked. Whenever the last vertex on p is adjacent to the start vertex s, the elementary path p is an elementary circuit. A similar algorithm with the same bound was given by Ehrenfeucht and Osterweil [20].

A more efficient algorithm with the time bound $O((n+e)(c+1))$ [21] was proposed by Johnson. The algorithm has two important strategies. The first one is that to avoid duplicating circuits, a vertex v must be blocked when it is added to some elementary path beginning in s, and keeps blocked as long as every path from v to s intersects the current elementary path at a vertex other than s; the other is that a vertex does not become a root vertex for constructing elementary paths unless it is the least vertex in at least one elementary circuit. These two strategies reduce much of the fruitless searching of previous algorithms, which results in the lower time bound.

There are some other algorithms for finding all the elementary circuits. For example, Weinblatt gave an algorithm similar to Tiernan's, but it has relatively large storage requirements [22]. Also based on the edge weight, Yamada and Kinoshita developed a heuristic algorithm to list all negative cycles [23].

The main difference between the algorithms above and our approach is as follows. Firstly, the algorithms above search for all elementary cycles directly from the directed graph, while our approach first computes accepting strongly connected components (ASCCs), and then searches for elementary accepting cycles of every ASCC, which avoids searching for a great many ineffective paths. Secondly, the algorithms above are based on internal memory, but our approach stores the whole state space and hash tables in external memory, which allows to handle larger-scale problem for finding all elementary cycles.

3 Preliminaries

In this section, for convenience, we give a brief introduction of some necessary notions about graphs, I/O complexity model, and minimal perfect hash function used in this paper. Please refer to [27], [28], [29], [30] for more details.

3.1 Related Definitions of Graphs

In this paper, we treat an automaton as a directed graph that has an initial state s_0 and an accepting state set $\mathcal{F}$. Sometimes we use the term *state* for node and *transition* for edge.

A directed graph G is a pair (V, E) where V is a set of nodes (or vertices), and $E \subseteq V \times V$ is a set of edges. A node u is said to be reachable from v in (V, E) if and only if there is a sequence of nodes $v_1, v_2 \ldots v_n$ such that $v = v_1$, $u = v_n$ and $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, n-1$. A path is a sequence of nodes $\rho_{vu} = (v = v_1, v_2, \ldots, v_k = u)$ such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. A cycle is a path in which the first and last nodes are identical. The graphs in our definitions contain no loop (edges of the form (v, v)) and no multiple edges between the same nodes, namely if edge $(u, v) \in E$, then $u \neq v$ and the edge (u, v) consists of unique directed arc from u to v. An accepting cycle is a cycle going through some accepting state. Previous I/O LTL model checking algorithms [2], [3], [4] work by finding an accepting cycle as a counterexample. In this paper, we aim at finding all accepting cycles.

Given a graph (V, E), a strongly connected component (SCC) is a maximal subgraph (V', E') such that $V' \subseteq V$, $E' = \{(u, v) \in E \,|\, u, v \in V'\}$, and u is reachable from v in (V', E') for all $u, v \in V'$. An accepting strongly connected component of the automaton is defined as an SCC containing some accepting state $s \in F$. Given a state s, according to the definition of ASCC, we know that if G_1 and G_2 are two ASCCs containing state s, then G_1 and G_2 are identical. We use $ASCC(s)$ to denote the ASCC containing state s.

3.2 I/O Complexity Model

As disk access is orders of magnitude slower than internal memory access [31], complexities of external memory algorithms are usually measured in terms of the number of I/O operations. For complexity analysis of external memory algorithms, a widely used model is the one by Aggarwal and Vitter [28]. In the model, the complexity of an I/O algorithm is further parameterized by N, M, and B, where N denotes the total number of items of a system stored on disk, M denotes the number of items that fit into the internal memory, and B denotes the number of items that can be transferred in a single I/O operation. Thus, the number of I/O operations for scanning N nodes on disk is $O(scan(N)) = O(N/B)$. As another important operation, sorting N nodes on disk has an I/O complexity of $O(sort(N)) = O(N/B \cdot \log_{M/B} N/B)$.

In this paper, we adopt this complexity model to analyze the I/O complexity of our approach.

3.3 The Minimal Perfect Hash Function

A minimal perfect hash function is a one-one correspondence between a static set Q and $\{0, \ldots, |Q|\}$, which requires that all elements in Q are distinct mutually. MPHF has many applications as other hash functions do, but with the advantage that no collision resolution has to be implemented. An asymptotically space optimal, practical algorithm for generating MPHFs was designed in [29], which has I/O complexity $O(sort(|Q|))$. The external memory variant, refereed to as external perfect hashing (EPH), was also given in [30].

In this paper, we use algorithm EPH to construct a minimal perfect hash function. From references [4] and [30], we know MPHF constructed by EPH can be stored in less than 4 bits of internal memory per state. Thus our approach can handle systems with more than $2 \cdot 2^{30} \cdot 8/4 = 2^{32}$ states on a computer with 2 GB RAM.

4 I/O Efficient Detection of All Accepting Cycles

Each counterexample can be represented as a pair of an accepting cycle and a path from the initial state to the cycle. When finding an accepting cycle, we easily find a path from the initial state to the cycle by using DFS. We say two counterexamples are equivalent if they have the same accepting cycles. Thus, the problem of detecting all (or a set of) counterexamples may be reduced to that of detecting all (or a set of) accepting cycles. In this section, we propose an I/O efficient approach for detecting all accepting cycles, namely DAAC.

4.1 Framework of DAAC

The basic framework of DAAC is as follows. To find all accepting cycles, DAAC proceeds by traversing the accepting state set. For each accepting state, DAAC first searches for the ASCC containing the accepting state by using the algorithm SFA (Search For ASCC), and then finds accepting cycles of the ASCC by the algorithm Find Accepting Cycles of the ASCC (FACA). When all accepting states are traversed, all accepting cycles of the system are found.

The above framework is based on the observation that given a graph with an accepting state set, all accepting cycles going through the same accepting state must be in the same ASCC, and all ASCCs are disjoint. Thus, finding all accepting cycles of a system amounts to finding accepting cycles in each ASCC.

According to the definition of ASCC, each state in an ASCC must be on some accepting cycle. Thus, compared to the naive method which directly finds accepting cycles from the graph, this framework enables our approach to avoid searching for a great many ineffective paths. For example, considering Fig. 1, suppose the graph has $2n$+4 states and only one accepting state s. Obviously, the left sub

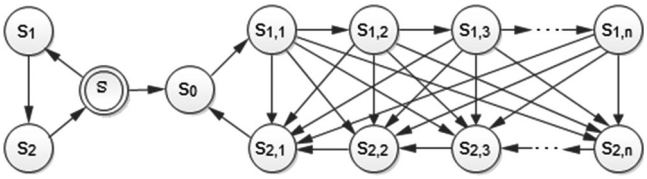


Fig. 1. Graph with one accepting cycle.

graph is one accepting cycle, and the right subgraph contains the following n^2 cycles (not accepting cycles):

$s_0 \rightarrow s_{1,1} \rightarrow s_{2,1} \rightarrow s_0$,
$s_0 \rightarrow s_{1,1} \rightarrow s_{2,2} \rightarrow s_{2,1} \rightarrow s_0$,
$\ldots$,
$s_0 \rightarrow s_{1,1} \rightarrow s_{2,n} \rightarrow s_{2,n-1} \rightarrow \ldots \rightarrow s_{2,1} \rightarrow s_0$,
$\ldots$,
$s_0 \rightarrow s_{1,1} \ldots \rightarrow s_{1,n} \rightarrow s_{2,1} \rightarrow s_0$,
$s_0 \rightarrow s_{1,1} \ldots \rightarrow s_{1,n} \rightarrow s_{2,2} \rightarrow s_{2,1} \rightarrow s_0$,
$\ldots$,
$s_0 \rightarrow s_{1,1} \ldots \rightarrow s_{1,n} \rightarrow s_{2,n} \rightarrow s_{2,n-1} \ldots \rightarrow s_{2,2} \rightarrow s_{2,1} \rightarrow s_0$.

Note that to find all accepting cycles, every state may be visited repeatedly. In our approach, we first compute ASCC by traversing the whole state space using BFS and MPHF, which needs to execute $2(2n+4) = 4n+8$ steps. Then, we find the accepting cycle with three steps. Thus our approach needs $4n+11$ steps in total. Previous approaches, such as Johnson's algorithm [21], search directly from the graph to find all accepting cycles, traversing not only the whole state space but also all possible paths, which needs to execute about $O(\{(1+2)+(1+3)+\cdots+(1+(n+1))\} \times n+3) = O(n^3/2+5n^2/2+3) = O(n^3)$ steps.

In order to further lower the I/O complexity and improve the performance of DAAC, we propose an intersection computation technique and a dynamic path management technique, and exploit a minimal perfect hash function. The intersection computation technique is to lower the I/O complexity and improves the performance of the algorithm SFA. The dynamic path management technique allows to reduce the memory dithering and so improves the performance of the algorithm FACA. With MPHF, our algorithm requires only one I/O operation to locate a record in a hash table on disk according to a hash value, which significantly improves the efficiency of DAAC.

In the following, we describe three main algorithms in DAAC: the top-level algorithm, SFA, and FACA. Note that all state tables are stored on disk and have the same data structure, consisting of one field "state"; all hash tables are stored on disk and have the same data structure, consisting of two fields: the hash field and the tag field. Initially every hash table has N different records, in which values of the hash field range from 1 to N and all values of the tag field are 0, and every hash table is sorted by the hash field, where N is the number of states of the system. In fact, values of the hash field of a hash table corresponds to hash values of N states of the system.

4.2 DAAC Algorithm: Top Level

We now describe the DAAC algorithm (see Algorithm 1). DAAC first enumerates all reachable states of the automaton using external breadth first search (BFS) by invoking the function $enumerateBFS()$, and then constructs a MPHF by using the algorithm EPH ($constructMPHF()$). After that, the algorithm finds all accepting cycles of the system according to the proposed framework above. To avoid repeated computation and speed up DAAC, after finding the ASCC containing an accepting state, DAAC deletes all accepting states contained in the ASCC from $\mathcal{F}$ by a *for* loop.

Algorithm 1. The Algorithm for Detecting All Accepting cycles

Input s_0: the initial state; $succ^+$: forward transition; $succ^-$: backward transition; $\mathcal{F}$: the set of accepting states;
Output All elementary accepting cycles;
Function $DAAC()$
Var $hash$: minimal perfect hash function; $tableH$: hash table; $tableS$: state table;
$tableS := enumerateBFS(s_0, succ^+)$;
$hash := constructMPHF(tableS)$;
while $\mathcal{F} \neq \emptyset$ **do**
  pick $s \in \mathcal{F}$;
  $clear(tableH)$;
  $tableH := SFA(s, succ^+, succ^-)$;
  /* $tableH$ corresponds to an ASCC */
  for all $t \in \mathcal{F}$ **do**
    if $gettag(tableH, hash(t)) = 1$ **then**
      $\mathcal{F} = \mathcal{F} \setminus \{t\}$;
    end if
  end for
  $FACA(tableH)$;
end while
EndFunction

The function $clear()$ is to reset a hash table by modifying all tags to 0. The function $gettag(tableH, hash(t))$ is to get the tag value corresponding to $hash(t)$ from the hash table $tableH$.

The algorithm $enumerateBFS()$ exploits an external BFS technique presented in Section 3.1 of literature [7], which uses external memory(disk) and delayed duplicate detection (DDD) [7], [32]. The usage of external memory is to avoid the limitation of internal memory for the algorithm, and DDD improves greatly the algorithm's efficiency. DDD is based on the observation that when a BFS is used to enumerate the space, it has to maintain a set of visited states to prevent their re-exploration. Since the graphs are large for large systems, the set of visited states cannot be completely kept in the main memory and should be stored on the external memory device. A newly generated state does not need to be checked against the state set immediately; one can postpone the checking until an entire level of the BFS has been explored and then check all states in the level together by linearly reading the set from the disk. Please refer to [7], [32] for more details.

4.3 Search for ASCC

Given an accepting state, the algorithm SFA is to compute the ASCC containing the accepting state. The underlying idea is based on the observation that every state in this ASCC is reachable from this accepting state, and this accepting state is also reachable from every state in this ASCC. Thus we first compute the forward reachable state set and backward reachable state set from this accepting state, and

then compute their intersection which is just the ASCC containing this accepting state.

In the computation process of reachable sets, we need to consider memory shortage problem for large-scale systems, and the problem of efficiency of duplicate detection of states, where duplicate detection of a state is to check if the state was searched for before. The first problem can be effectively solved by storing the state tables and hash tables on disk. The second problem can be effectively solved by using hash tables on disk. This is because hash values ranging from 1 to N in a hash table are sorted, and locating a state in the hash table needs only one I/O operation, where N is the number of states of the system. Thus the duplicate detection of a state costs only one I/O operation.

Computing intersection of two state sets is another key factor that affects the performance and the I/O complexity of the ASCC computation. Generally, intersection computation is with non-linear worst-case time complexity. To the best of our knowledge, the best algorithm for set intersection [33] has a linear expected time complexity. In this section, based on a minimum perfect hash function (MPHF), we propose an I/O efficient algorithm for computing set intersection with linear worst-case I/O complexity, which lowers the I/O complexity and improves the performance of SFA. The basic idea of this algorithm is to translate the intersection computation problem of two state sets into the tag comparison problem of two hash tables.

In this section, we first propose an algorithm for the intersection computation of two state sets, and then introduce an algorithm for computation of reachable sets, finally, based on the two techniques above, we propose an I/O efficient algorithm of searching for ASCC, namely SFA.

4.3.1 Computing Intersection of Two State Sets

We first define an equivalent representation of a state set. Suppose Q is a state set, then an equivalent representation of Q is a hash table H which satisfies the condition that $t \in Q$ iff the entry that contains $hash(t)$ in H has tag value 1. Thus, we can translate the problem of computation of intersection of two state sets into the problem of tag checking of two hash tables.

Algorithm 2. The Algorithm for Computing Intersection of Two State Sets

Input H_1, H_2: hash table;
Output H_1: hash table;
Function $intersect()$
Var t: state;
for all $hash(t) \in H_1$ **do**
  if $gettag(H_1, hash(t)) = 1$ **then**
    if $gettag(H_2, hash(t)) = 0$ **then**
      $settag(H_1, hash(t), 0)$;
    end if
  end if
end for
$Return(H_1)$;
EndFunction

Suppose Q_1 and Q_2 are two state sets, and H_1 and H_2 are two hash tables equivalent to Q_1 and Q_2, respectively. Then, the set of states whose tag values are 1 in two hash tables is the intersection of two state sets. We store the final result of the algorithm in H_1. The function $settag(H_1, hash(t), *)$ is to set the tag value corresponding to $hash(t)$ in H_1 to $*$. The computation process of intersection of two state sets is described in Algorithm 2.

4.3.2 Search for the Reachable State Set of an Accepting State

We start with Algorithm 3 that computes the sets of states reachable from an accepting state $s \in \mathcal{F}$, including the forward reachable set and the backward reachable set. Let **I-QUEUE** be type of internal queue, and **E-QUEUE** be type of external queue. We assume the basic operations over a queue data structure, such as $enqueue()$ that inserts an element at the end of a queue and $dequeue()$ that pops an element from the beginning of a queue. An empty queue is represented by ϵ. Moreover, we assume internal queue has a capacity of m elements, bounded by the limited internal memory. Note that $succ$ may be $succ^+$ and $succ^-$, which expresses forward transition and backward transition, respectively.

Algorithm 3. The Algorithm for Computing the Reachable Set of an Accepting State

Input s: accepting state; $succ$: transition;
Output H: hash table;
Function $CRS()$
Var = $current$, $next$: **I-QUEUE**; $currentTable$, $nextTable$: **E-QUEUE**;
  $currentTable := \epsilon$;
$currentTable.enqueue(s)$;
$clear(H)$;
do
$current := \epsilon$; $next := \epsilon$; $nextTable := \epsilon$;
while $currentTable \neq \epsilon$ **do**
  repeat
    $current.enqueue(currentTable.dequeue())$;
  until $\#(current) = m$ or $\#(currentTable) = 0$
  while $current \neq \epsilon$ **do**
    $t := current.dequeue()$;
    for all $u \in succ(t)$ **do**
      /*$succ(t)$ is stored as list of all successors of t*/
      if $gettag(H, hash(u)) = 0$ **then**
        $next.enqueue(u)$;
        if $\#(next) = m$ **then**
          repeat
            $nextTable.enqueue(next.dequeue())$;
          until $next = \epsilon$
        end if
        $settag(H, hash(u), 1)$;
        /*Show u has been visited*/
      end if
    end for
  end while
end while
repeat
  $nextTable.enqueue(next.dequeue())$;
until $next = \epsilon$
$currentTable := nextTable$;
while $nextTable \neq \epsilon$
$Return(H)$;
EndFunction

Starting from an accepting state s, the algorithm searches level by level for all reachable states from s. The $do\text{-}while$ loop creates all levels. The outer $while$ loop is to create the next level from the current level which is stored in $currentTable$. During every outer $while$ loop, the algorithm first moves $min(m, \#(currentTable))$ states to the queue $current$ from the current level by a $repeat$ loop, and then computes all non-duplicate successors of every state in the queue $current$ and stores them in the queue $next$, and sets the tag values of these non-duplicate states as 1 in H, where $\#(currentTable)$ denotes the number of states of $currentTable$ and $min(m, \#(currentTable))$ expresses the smaller one between m and $\#(currentTable)$. The duplicate detection of a state is carried out by calling the function $gettag()$ to check if its tag value is 1.Thus the duplicate detection of a state needs only one I/O operation. If the queue $next$ is full, then the algorithm moves all states in $next$ to $nextTable$ on disk. When the queue $current$ has been handled, the algorithm moves $min(m, \#(currentTable))$ states to the queue from the current level again, and so on until the whole current level has been handled, namely $currentTable$ is empty.

Finally, the algorithm outputs H in which only the tag values of all states reachable from the accepting state s are 1.

4.3.3 Computing ASCC

We now consider the algorithm SFA. Based on the idea above, the algorithm for the computation of set intersection and the algorithm for computation of reachable set, given an accepting state s, Algorithm 4 can efficiently compute $ASCC(s)$. The result H returned by function $SFA()$ is a hash table corresponding to $ASCC(s)$.

Algorithm 4. The Algorithm of Search for ASCC

Input s: accepting state; $succ^+$: forward transition; $succ^-$: backward transition;
Output H: hash table;
Function $SFA()$
Var H_1, H_2: hash table;
$clear(H_1)$;
$clear(H_2)$;
$H_1 := CRS(s, succ^+)$;
$H_2 := CRS(s, succ^-)$;
$H := intersect(H_1, H_2)$;
$Return(H)$;
EndFunction

4.4 Search for all Accepting Cycles in an ASCC

After attaining an ASCC using the algorithm SFA, we need to find all accepting cycles in this ASCC. This is accomplished by the algorithm FACA, which is an extension of Johnson's algorithm [21] to external memory and our framework. In the algorithm FACA, the whole ASCC is stored in external memory by a hash table, and search path is first stored in a stack in internal memory. We need to efficiently manage the stack because there exists the memory dithering problem when using stack. In this section, we first propose a dynamic path management technique denoted by Dynamic Path Management, and then design the algorithm FACA by combining DPM, MPHF and Johnson's algorithm [21].

4.4.1 Equivalence of Accepting Cycles

We first give some definitions about accepting cycles and an equivalence proposition.

Definition 1. *A state s is said to be elementary for an accepting cycle C if there is no accepting subcycle of C after s is deleted from C; otherwise, s is said to non-elementary.*

Definition 2. *An accepting cycle C is said to be elementary if every state on C is elementary.*

Definition 3. *Two accepting cycles are said to be equivalent if they have the same elementary states.*

According to elementary graph theory, we prove an important claim: Any accepting cycle C is equivalent to some elementary accepting cycle.

Proof. Assume the accepting cycle C goes through the accepting state s, and t is a state appearing twice on C. Obviously, after deleting the sequence of states between the first t and the second t, the remaining sequence of states still constructs an accepting cycle going through s. We deal with all duplicate states like this and finally attain an accepting cycle D without duplicate states. This accepting cycle is obviously an elementary accepting cycle and equivalent to C. □

With the definitions and the claim above, we have that, in order to find all accepting cycles, it is sufficient to find all elementary accepting cycles. This can significantly reduce number of counterexamples to be searched for, but not impair the work that uses multiple counterexamples for better debugging.

4.4.2 Dynamic Path Management

When an elementary accepting cycle is too long to be stored in the internal memory, the search path management has a big impact on the performance of DAAC. In this section, we propose a dynamic path management technique, namely DPM. In this algorithm, we employ a stack in internal memory and a state table $tableP$ in external memory to store the whole search path.

During the search, when the stack is full and a new state is generated, we move states from $stack$ to $tableP$ in order to avoid memory overflow. However, this may result in the phenomenon that states are frequently moved inside and outside internal memory.

In the following, we analyze how this phenomenon occurs. Suppose that we swap M states at a time between internal memory and disk, where M is the number of states of the stack when the stack is full. When $stack$ is full, if we transfer M states from $stack$ to disk, then $stack$ becomes empty. In succession, if the algorithm needs to pop a state from $stack$ because of backtracking, then immediately those M states just moved to disk have to be moved back to internal memory, and thus $stack$ becomes full. Afterwards, if another new state is generated and needs to be pushed into $stack$, then the M states have to be moved outside internal memory again to make space for the new state, and so on. We call such a phenomenon of frequent movement of state block as memory dithering, which significantly increases disk accesses and thus the algorithm's I/O complexity.

In order to reduce memory dithering, we propose an efficient technique which works as follows.

1) When $stack$ is full, the algorithm moves only some states from $stack$ into $tableP$ to make memory space for new states. To be more detailed, because stack operates in First In, Last Out (FILO), the algorithm moves k states in the bottom of the stack, which will not be used temporarily, into $tableP$, and releases the corresponding memory space, and then the bottom pointer of $stack$ points to the $(k+1)$-th state, where $k = \#(stack) \cdot \rho$ and ρ is a parameter that $0 < \rho < 1$. This procedure is implemented by the function $Mem\text{-}Disk()$.
2) When $stack$ becomes empty, if $tableP$ is not empty, then the algorithm pushes k states stored most recently in $tableP$ into $stack$ in turn and deletes them from $tableP$, where $k = min(\#(tableP), M \cdot \rho)$. The corresponding procedure is implemented by the function $Disk\text{-}Mem()$.

By reserving some states in the stack when $stack$ is full, we ensure there are always some states in the stack, which to some extent reduces the memory dithering phenomenon.

4.4.3 Finding All Elementary Accepting Cycles in an ASCC

Given an ASCC described by a hash table H, the algorithm FACA is to compute all elementary accepting cycles in the ASCC. It is mainly composed of three functions $unblock()$, $dfs()$ and $FACA()$. Note that FACA uses the same MPHF as SFA.

The algorithm FACA first marks every element of the array $ASCCof$ according to H by calling the function $mark()$, where $ASCCof$ is a global internal bit array with size N. That is, for every state t, if the tag of $hash(t)$ in H is 1, then $ASCCof[hash(t)]$=1; otherwise, $ASCCof[hash(t)]$=0. Thus, the array $ASCCof$ expresses an ASCC. Assume $s_1, s_2, \ldots, s_m$ are all accepting states in the ASCC. FACA searches for all elementary accepting cycles in the ASCC by traversing the m accepting states.

In the ith traverse (namely the ith $while$ loop, $1 \le i \le m$), FACA searches for all elementary accepting cycles going through s_i by recursively calling the function $dfs()$. In the search process, the successors of every state are implicitly generated by the function $successor()$. Note that here we only consider the successors in the ASCC, namely all successors t such that $ASCCof[hash(t)]$=1. The states on current search path are kept on $stack$. If a successor of some state is just s_i, then FACA finds an elementary accepting cycle going through s_i, which is a sequence of states stored successively in $tableP$ and $stack$, and FACA outputs the cycle, and then continues to search for other elementary accepting cycles going through s_i. When the stack is full, FACA invokes the function $Mem\text{-}Disk()$ to manage the stack. To avoid state duplication on the same accepting cycle, a state u is blocked by setting $blocked(u)$ to be true when it is added to the current search path beginning in s_i. And it stays blocked as long as every path from u to s_i intersects the current path at a state other than s_i. This reduces much of the fruitless searching. To avoid duplicate searching for elementary accepting cycles going through s_i, the algorithm deletes s_i from the ASCC by setting $ASCCof[hash(s_i)]$=0 after finding all elementary accepting cycles going through s_i. FACA is described by Algorithm 5.

Algorithm 5. The Algorithm for Finding Accepting Cycles in an ASCC

```
Var s: state; ASCC of: internal bit array with size N;
Function unblock(t: state )
blocked(t):=false;
for w ∈ B(t) do
  delete w from B(t);
  if blocked(w) = true then unblock(w) endif
end for
EndFunction
logical Function dfs(u: state)
Var t: state; flag: boolean variable;
if stack is full then Mem-Disk() endif
flag:=false; push u into stack; blocked(u):= true;
for all t ∈ successor(u) do
  if ASCCof[hash(t)]=1 then
    /*namely, t is a successor of u in the ASCC */
    if t = s then
      output the found cycle going through s;
      flag := true;
    else
      if blocked(t) = false then
        if dfs(t) then flag := true; end if
      end if
    end if
  end if
end for
if flag = true then unblock(u)
else
for all t ∈ successor(u) do
  if ASCCof[hash(t)]=1 and u ∉ B(t) then
    put u in B(t);
  end if
end for
endif
pop u from stack; dfs := flag;
if stack is empty and tableP is not empty then
  Disk-Mem();
end if
EndFunction
Function FACA(H: hash table)
/*H corresponds to an ASCC */
Var i, m: integer; t: state;
mark(ASCCof, H); i := 1;
/* Let {s_i|1 ≤ i ≤ m} be the set of all accepting states in the
ASCC*/;
while i ≤ m do
  for all t such that ASCCof[hash(t)] = 1 do
    blocked(t) := false; B(t) := ∅;
  end for
  empty(stack); s := s_i;
  dfs(s_i);
  ASCCof[hash(s_i)]=0; /*delete s_i from ASCC*/
  i := i + 1;
end while
EndFunction
```

Note that when the number of accepting cycles is very large, we may consider to set a bound k on the number of

accepting cycles in advance. In search process, when the number of accepting cycles reaches the bound k, the approach terminates.

5 Correctness of DAAC

In this section, we give the proof for the correctness of DAAC. The proof consists of two parts. One is for the correctness of algorithm SFA which finds accepting strongly connected components; the other for that of algorithm FACA which finds all elementary accepting cycles of some ASCC.

5.1 Correctness of SFA

Lemma 1. *The algorithm SFA returns an ASCC.*

Proof. Because there is a one to one correspondence between the state set of a system and the natural number set $\{1, \ldots, N\}$, where N is the number of states of a system, we consider a hash value as a state. It is sufficient to prove that the hash table, which is returned by the function $SFA()$ in Algorithm 4, expresses an ASCC.

The statement H_1:=$CRS(s, succ^+)$ means that every state in the hash table H_1 is reachable from the accepting state s, while the statement H_2:=$CRS(s, succ^-)$ means that s is reachable from every state in the hash table H_2. Thus, the intersection H of H_1 and H_2 constitute an ASCC containing accepting state s. □

Lemma 2. *Any accepting strongly connected component can be attained by calling the function $SFA()$.*

Proof. It is sufficient to prove that given any accepting state s, the ASCC containing s (namely $ASCC(s)$) can be obtained by the function $SFA()$.

We branch our proof into two cases according to whether the function $SFA(s, succ^+, succ^-)$ is invoked in $while$ loop in Algorithm 1.

1) The case that $SFA(s, succ^+, succ^-)$ is invoked. From Lemma 1, we have that the function $SFA(s, succ^+, succ^-)$ returns an ASCC containing the accepting state s, namely $ASCC(s)$. Thus the lemma holds for this case.
2) The case that $SFA(s, succ^+, succ^-)$ is not invoked. By Algorithm 1, we have that there exists an accepting state u such that the function $SFA(u, succ^+, succ^-)$ is called in $while$ loop and s is in $ASCC(u)$. Clearly, $ASCC(s)$ is the same as $ASCC(u)$. From the first case, we have that $ASCC(u)$ can be attained by calling the function $SFA()$. Thus $ASCC(s)$ can also be attained.

This completes the proof. □

5.2 Correctness of FACA

The correctness of the algorithm for finding all accepting cycles of some ASCC can be proved along similar line as [21].

Lemma 3. *In FACA, for any state u, if there is a path from the state to s which intersects the stack only at s, then $blocked(u)$ is false.*

Proof. Suppose the path is $(u = v_k, v_{k-1}, \ldots, v_1, s)$. Assume that $blocked(v_k)$ is true. Then u must be the state which was added to the stack earlier and was subsequently popped from stack. Certainly v_1, v_2, …, v_{k-1} also pop from the stack successively before that. Because s is a successor of v_1, the $flag$ is true before v_1 pops from the stack. It follows that the $unblocked(v_1)$ is called, and $blocked(v_1)$ is set false, and $dfs(v_1)$ returns true. Because $dfs(v_1)$ returns true, the $flag$ is also true before v_2 pops from the stack. It follows that the $unblocked(v_2)$ is called, and $blocked(v_2)$ is set false, and $dfs(v_2)$ returns true, and so on. Finally the $unblocked(v_k)$ is called, and $blocked(v_k)$ is set false, which contradicts with assumption. Thus, we conclude that $blocked(u)$ is false. □

Lemma 4. *In the function $dfs()$, for any state $x \neq s$, there is a call $unblock(u)$ which sets $blocked(x)$ false if and only if*

1) *there is a path, containing u, from x to s on which only u and s are on the stack;*
2) *there is no path from x to s on which only s is on the stack.*

Proof. The main difference between $dfs()$ of DAAC and $CIRCUIT()$ of Johnson's algorithm [21] is that the graph (or stack) is partly put on disk in $dfs()$, which does not change the structure of the graph (or stack), namely, its vertex set and edge set do not change. Thus, we can prove the lemma in the same way as [21, Lemma 1]. □

By Lemma 4, we easily get the following lemma.

Lemma 5. *Every state on the stack remains blocked.*

Theorem 4. *The algorithm outputs only elementary accepting cycles.*

Proof. Because s is an accepting state in the algorithm, by the function $dfs(s)$, we know that only accepting cycles are output. By Lemma 5, every state on the stack remains blocked. Thus no state can be repeated on the stack. Therefore, the algorithm outputs only elementary accepting cycles. □

Theorem 5. *Every elementary accepting cycle is output once and only once.*

Proof. Let $(s = v_1, v_2, \ldots, v_k, v_1)$ be any elementary accepting cycle which s is an accepting state. Then, we conclude that $(v_1, v_2, \ldots, v_k, v_1)$ is in some ASCC. Because s is an accepting state, a first call $dfs(s)$ will eventually occur and s is the first state in the stack. By Lemma 3, v_2 is not in the blocked situation. Thus, the stack will later be $(s = v_1, v_2)$. By induction and using Lemma 3, whenever the stack is $(s = v_l, v_2, \ldots, v_i)$, for $i < k$, the stack will later be $(s = v_l, v_2, \ldots, v_{i+1})$. Thus, DAAC output $(s = v_1, v_2, \ldots, v_k, v_1)$.

In the function $FACA()$, for $i < m$, the algorithm deletes s_i from ASCC after it executes $dfs(s_i)$, and then executes $dfs(s_{i+1})$. Thus, for $i \neq j$, the elementary accepting cycles generated by $dfs(s_i)$ are different from that by $dfs(s_j)$. Moreover, in the call $dfs(s_i)$, no elementary accepting cycle is output more than once, since for any stack $(s_i = v_1, v_2, \ldots, v_k)$ with v_k on top, once v_k is

removed the same stack cannot reoccur. Therefore, every elementary accepting cycle is output only once. □

6 Complexity Analysis

In this section, we apply the model of Aggarwal and Vitter to analyze the I/O complexity of DAAC, and compare DACC with DAC, MAP and IDDFS in terms of I/O complexity.

6.1 The I/O Complexity of DAAC

The I/O complexity of DAAC directly depends on that of algorithms SFA and FACA.

Lemma 6. *The I/O complexity of the algorithm for computation of intersection of two state sets is $O(\mathtt{scan(N)})$, where N is the number of states in an automaton.*

Proof. Because H_1 and H_2 in Algorithm 2 are two hash tables sorted by hash values, locating a record in H_1 or H_2 according to the hash value of a state needs only one I/O operation. Thus, $settag()$ and $gettag()$ need one I/O operation, respectively. It follows that the I/O complexity of the algorithm is $O(\mathtt{scan(N)})$, linear in size of the state space of the automaton. □

Lemma 7. *The I/O complexity of the algorithm for computation of reachable state set is $O(\mathtt{scan(N)})$.*

Proof. The main factors impacting I/O complexity of Algorithm 3 is three **repeat** statements, $gettag()$ and $settag()$. The accumulated effect of the first **repeat** statement in all iterations is equivalent to moving N states from disk to the internal memory. The accumulated effect of the other two **repeat** statements is equivalent to moving N states from the internal memory to disk. Thus, these three **repeat** statements take $(2 \cdot N/B)$ I/O operations in all iterations in total. The functions $gettag()$ and $settag()$ are executed N times, respectively, which takes $2N$ I/O operations in total. Therefore, the I/O complexity of the algorithm is $O(\mathtt{scan(N)})$. □

Theorem 6. *The I/O complexity of the algorithm SFA is $O(\mathtt{scan(N)})$.*

Proof. By Lemmas 6 and 7, clearly the I/O complexity of the algorithm SFA is $O(\mathtt{scan(N)})$. □

Lemma 8. *The search for any elementary accepting cycle in an ASCC requires $O(scan(N'))$ I/O operations, where N' is the number of states of the ASCC.*

Proof. By the proof of [21, Lemma 3], we know no state can be unblocked more than once unless an elementary accepting cycle is output. Thus, in the search process of any elementary accepting cycle C in the ASCC, the worst case is that all states in the ASCC are traversed twice and there is the most times of movement of state block from the internal memory to disk or from disk to the internal memory. Thus, when $stack$ is full, FACA moves $(M \cdot \rho)$ states from $stack$ to $tableP$, and then pops $(M \cdot (1-\rho))$ states one by one from $stack$, and $stack$ becomes empty, where M is the number of states of the stack when the stack is full; afterwards, FACA transfers $(M \cdot \rho)$ states to $stack$ from $tableP$ again, and then generates and pushes $(M \cdot (1-\rho))$ states one by one into $stack$, and $stack$ becomes full; and goes on like this until all states in the ASCC are traversed. In this process, the block of states is moved $((N' - M \cdot \rho)/(M \cdot (1-\rho)) - 1)$ times from $stack$ to $tableP$. Because every movement of the block of states needs $(M \cdot \rho/B)$ I/O operations and the number of states moved from $stack$ to $tableP$ is equal to that from $tableP$ to $stack$, the whole process costs $2((N' - M \cdot \rho)/\ (M \cdot (1-\rho)) - 1) \cdot (M \cdot \rho/B)$ I/O operations. Therefore, the search for any elementary accepting cycle in the ASCC needs $O(\rho/(1-\rho) \cdot scan(N'))$ I/O operations, namely $O(scan(N'))$ I/O operations. □

Theorem 7. *The I/O complexity of the algorithm FACA is $O(c' \cdot scan(N))$, where c' is the number of all elementary accepting cycles contained in the ASCC and N is the number of states of the system.*

Proof. Suppose that N' is the number of states of the ASCC. From Algorithm 5, we know that all operations are executed in internal memory and need no I/O operation except for the functions $Mem\text{-}Disk()$, $Disk\text{-}Mem()$ and $mark()$. The function $mark()$ costs N I/O operations. By Lemma 8, it needs to take $O(c' \cdot scan(N'))$ I/O operations to find all accepting cycles in the ASCC. Thus the I/O complexity of the algorithm FACA is $O(c' \cdot scan(N') + N)$, namely $O(c' \cdot scan(N))$. □

Theorem 8. *The I/O complexity of the approach DAAC is $O((\ell + c) \cdot \mathtt{scan(N)} + \mathtt{sort(|E|)})$, where $\ell = max\{\delta(s,u) - \delta(s,v) + 1 \mid (u,v) \in E\}$ and c is the number of all elementary accepting cycles of an automaton.*

Proof. Let L be the number of all ASCCs in an automaton, and c_i be the number of all elementary accepting cycles of the ASCC in the ith $while$ loop in the Algorithm 1$(i = 1, \ldots, L)$.

In the Algorithm 1, enumerating all reachable states using the function $enumerateBFS()$ needs to perform $O(\ell \cdot \mathtt{scan(N)} + \mathtt{sort(|E|)})$ I/O operations [4], [7], [32], where $\ell = max\{\delta(s,u) - \delta(s,v) + 1 \mid (u,v) \in E\}$ is the length of the longest back-edge in the BFS graph or its locality [7]. Constructing the minimal perfect hash function MPHF by using the algorithm EPH costs $O(sort(N))$ I/O operations.

Executing the for loop needs $O(\mathtt{scan(|F|)})$ I/O operations. Thus, by Theorems 6 and 7, we have that executing the ith $while$ loop needs $O(\mathtt{scan(N)} + \mathtt{scan(|F|)} + \mathtt{c_i} \cdot \mathtt{scan(N)})$, namely $O(c_i \cdot \mathtt{scan(N)})$ I/O operations. Therefore, DAAC has the I/O complexity of $O(\ell \cdot \mathtt{scan(N)} + \mathtt{sort(|E|)} + \mathtt{sort(N)} + ((\Sigma_{i=1}^{L} c_i) \cdot \mathtt{scan(N)}))$, namely $O((\ell + c) \cdot \mathtt{scan(N)} + \mathtt{sort(|E|)})$. □

6.2 Complexity Comparison

We compare the I/O complexity of DAAC with that of algorithms DAC, MAP and IDDFS according to Theorem 8 and existing results concerning the I/O complexity of these algorithms.

Theorem 1 of [2] claims that the I/O complexity of DAC is $O(l_{SCC} \cdot (h_{BFS} + |p_{max}| + |E|/M) \cdot \mathtt{scan(N)})$, where p_{max} is the length of the longest path in the graph going through a trivial SCC (without self-loops), h_{BFS} is the BFS height, and

TABLE 1
Time Consumption of the Algorithms

		DAAC		DAC		MAP		IDDFS	
benchmark	size	time	cycles	time	cycles	time	cycles	time	cycles
Models with valid property									
$Elev.2(16)$,P4	173, 916, 122	04:32:11	0	06:15:31	0	06:43:12	0	07:17:24	0
$MSC(5)$,P4	119, 663, 657	02:13:35	0	02:25:15	0	02:47:28	0	02:37:42	0
$Phils(16,1)$,P3	61, 230, 206	02:15:32	0	01:50:51	0	01:39:46	0	02:03:46	0
$Lamport(5)$,P4	74, 413, 141	01:55:52	0	02:36:33	0	02:54:16	0	02:44:54	0
$Peterson(6)$,P4	4, 187, 461, 216	63:13:21	0	85:26:13	0	80:38:31	0	cannot handle	
$Szyman.(6)$,P4	6, 521, 314, 436	70:31:15	0	82:33:51	0	95:32:35	0	cannot handle	
Models with invalid property									
$Bakery(5,5)$,P3	506, 246, 410	01:57:21	18246	10:02:21	1	00:07:45	1	00:07:06	1
$Szyman.(4)$,P2	4, 555, 287	00:29:41	4982	00:37:12	1	00:03:52	1	00:03:02	1
$Elevator2(7)$,P5	43, 776	00:00:12	104	00:00:31	1	00:00:17	1	00:00:16	1
$Lifts(7)$,P4	73, 473	00:00:10	120	00:00:14	1	00:00:31	1	00:02:12	1

$|E|$ is the number of edges in the graph. Since $O(\texttt{sort}(|\texttt{E}|)) = O(|E|/B \cdot \log_{M/B}(|E|/B))$ and $|E| \leq N^2$, $O(\texttt{sort}(|\texttt{E}|))$ is smaller than $O(2|E|/B \cdot \log_{M/B}(N))$. We know $O(\log_{M/B}(N))$ is much lower than $scan(N)$. It follows that $O(\texttt{sort}(|\texttt{E}|))$ is much lower than $O(|E| \cdot \texttt{scan}(\texttt{N}))$. Therefore, DAAC is far superior to DAC from the I/O complexity viewpoint.

Regarding MAP, its I/O complexity is $O(|\mathcal{F}| \cdot ((d + |E|/M + |\mathcal{F}|)\texttt{scan}(\texttt{N}) + \texttt{sort}(\texttt{N})))$ in the case for candidate set in RAM, and $O(|\mathcal{F}|((d + |\mathcal{F}|)\texttt{scan}(\texttt{N}) + \texttt{sort}(|\mathcal{F}| \cdot |\texttt{E}|)))$ in the case for candidate set on disk, where d is the diameter of a graph [3], [5]. The I/O complexity of the first case is affected mainly by $O(|\mathcal{F}| \cdot |E|/M \cdot \texttt{scan}(\texttt{N}))$. From the above, we know that $O(\texttt{sort}(|\texttt{E}|))$ is much lower than $O(|E| \cdot scan(N))$. Thus, $O(\texttt{sort}(|\texttt{E}|))$ is much lower than $O(|\mathcal{F}| \cdot |E|/M \cdot \texttt{scan}(\texttt{N}))$. Generally N is less than $|E|$. Thus, $O(\texttt{sort}(\texttt{N}))$ is lower than $O(\texttt{sort}(|\texttt{E}|))$. It follows that, for the first case, DAAC outperforms MAP in terms of I/O complexity. In addition, it is obvious that $O(\texttt{sort}(|\texttt{E}|))$ is lower than $O(\texttt{sort}(|\mathcal{F}| \cdot |\texttt{E}|))$. Thus, for the second case, DAAC also outperforms MAP in terms of I/O complexity.

The I/O complexity of IDDFS is $O(\in_s \cdot \texttt{sort}(\texttt{N}) + \ell \cdot \texttt{scan}(\texttt{N}) + \texttt{sort}(|\texttt{E}|))$, where $\in_s = max\{\delta(s,v) \mid v \in V\}$ and $\ell = max\{\delta(s,u) - \delta(s,v) + 1 \mid (u,v) \in E\}$ is the length of the longest back-edge in the BFS graph or its locality [4]. It is obvious that $O(\texttt{scan}(\texttt{N}))$ is lower than $O(sort(N))$. Thus, we can observe that DAAC has lower I/O complexity than IDDFS.

7 Experiment

In this section, we demonstrate by experiments that finding all accepting cycles can be nearly as I/O efficient as searching for one. To do so, we compare runtime and allocated disk space of DAAC with that of DAC, MAP and IDDFS. The experimental results confirm the efficiency of DAAC for finding all accepting cycles.

7.1 Benchmarks

In this experiment, we selected the same benchmarks as [2] and [4] except for models $Peterson(6), P4$ and $Szyman.(6), P4$. The two models are to show the limitation of scales of systems IDDFS can verify. All selected benchmarks are from the BEEM project [34], which include models with valid properties and those with invalid properties, and range from less than 50,000 states to more than 6,000,000,000 states. They are typical ones in the literatures and serve as a good test bed to justify the efficiency and performance of model checking algorithms.

The explanation about the notations in the names of benchmarks in this experiment is as follows. Please refer to [34] for more details. $Elev2(16), P4$ is an elevator model with 16 levels and the fourth property to be verified. $MSC(5), P4$ is a queue lock mutual exclusion model (or algorithm) with five processes and the fourth property to be verified. $Phils(16,1), P3$ is a philosopher dining model with 16 philosophers, one plate and the third property to be verified. $Lamport(5), P4$ is a Lamport's fast mutual exclusion model with five processes and the fourth property to be verified. $Peterson(6), P4$ is a Peterson's mutual exclusion protocol with six processes and the fourth property to be verified. $Szyman.(6), P4$ is a Szymanski mutual exclusion protocol with six processes and the fourth property to be verified. $Bakery(5,5), P3$ is a Bakery mutual exclusion algorithm with five processes, five critical sectionS and the third property to be verified. $Lifts(7), P4$ is a distributed model for lifting trucks with seven lifts and the fourth property to be verified.

7.2 Experimental Setup

The four algorithms have been implemented on top of the DiVine library [35], providing the state space generator, and the STXXL library [36], providing the I/O primitives. In the experiments, for DAAC, we set the parameters ρ as 0.94 and no bound to the number of counterexamples.

All experiments were run on a PC with CPU $P4$ 2.4 G, memory 2 G, disk space 400 GB, and Ubuntu Linux 9.0 operation system. For each instance, each algorithm is performed for 10 runs. For each algorithm on each instance, we report the averaged runtime ("$time$") and averaged disk consumption ("$disk$"). The time format is $hh{:}mm{:}ss$ (the elapsed hours, minutes and seconds) which is the same as that of [2], and "01:01:01" means that an algorithm costs 1 hour and 1 minute and 1 second. "szie" is the number of states of an instance, and "$cycles$" is the number of the accepting cycles detected by an algorithm.

7.3 Experimental Results

The experimental results are listed in Tables 1 and 2. On models with invalid properties, our approach finds all accepting cycles and the other three approaches find only one accepting cycle. Even so, our approach is still superior to DAC. As shown in Table 1, for all four instances with

TABLE 2
Disk Space Consumption of the Algorithms

		DAAC		DAC		MAP		IDDFS	
benchmark	size	disk	cycles	disk	cycles	disk	cycles	disk	cycles
Models with valid property									
$Elev.2(16)$,P4	173, 916, 122	7.75GB	0	8.21GB	0	12.53GB	0	7.92GB	0
$MSC(5)$,P4	119, 663, 657	4.80GB	0	7.31GB	0	8.35GB	0	6.49GB	0
$Phils(16, 1)$,P3	61, 230, 206	2.82GB	0	4.23GB	0	6.42GB	0	6.15GB	0
$Lamport(5)$,P4	74, 413, 141	2.47GB	0	3.23GB	0	5.53GB	0	2.82GB	0
$Peterson(6)$,P4	4, 187, 461, 216	78.21GB	0	174.36GB	0	100.03GB	0	cannot handle	
$Szyman.(6)$,P4	6, 521, 314, 436	134.72GB	0	241.43GB	0	225.31GB	0	cannot handle	
Models with invalid property									
$Bakery(5, 5)$,P3	506, 246, 410	993MB	18246	26.16GB	1	432MB	1	341MB	1
$Szyman.(4)$,P2	4, 555, 287	792MB	4982	1.06GB	1	266MB	1	235MB	1
$Elevator2(7)$,P5	43, 776	152MB	104	441MB	1	170MB	1	123MB	1
$Lifts(7)$,P4	73, 473	70MB	120	516MB	1	85MB	1	87MB	1

invalid properties, DAAC is much faster than DAC on three instances. In addition, our approach is also comparable with MAP and IDDFS. From Table 1, we can observe that for two instances $Elevator2(7)$,P5 and $Lifts(7)$,P4 with the small number of elementary accepting cycles, our approach is as fast as MAP and IDDFS, but for the two instances $Bakery(5, 5)$,P3 and $Szyman.(4)$,P2 with the large number of elementary accepting cycles, our approach performs a bit slower than MAP and IDDFS. However, this is worthwhile because our approach can find all accepting cycles in a run.

On models with valid properties, as a by-product, our approach also outperforms DAC, MAP and IDDFS. Table 1 shows that DAAC is faster than its competitors on almost all models with valid properties, especially those large-scale models such as $Peterson(6)$, P4 and $Szyman.(6)$, P4. The main cause is that for models with valid properties, DAC, MAP and IDDFS need to create the whole state space eventually as DAAC does, but DAAC exploits some efficient techniques such as the dynamic path management technique, the efficient framework and the intersection computation technique, which greatly improve its performance. Additionally, DAAC can handle the benchmarks $Peterson(6)$, P4 and $Szyman.(6)$, P4, which are too large for IDDFS. This is because IDDFS exploits heuristic EPH to construct a minimal perfect hash function, which needs 5 bits of internal memory for every state. Therefore, IDDFS cannot handle systems with more than $2 \cdot 2^{30} \cdot 8/5$ states on a computer with 2 GB RAM [4]. Moreover, the space consumption of DAAC is less than that of DAC, MAP and IDDFS on almost all models with valid properties.

In summary, DAAC has better practical efficiency on the whole than DAC, MAP and IDDFS.

Similar to IDDFS, DAAC also has the problem that the scales of the verified systems are to some extent limited because it exploits a minimal perfect hash function. But generally DAAC can handle larger-scale systems than IDDFS. This is mainly because DAAC uses EPH to construct a minimal perfect hash function and needs less than 4 bits of internal memory for every state. Comparatively, IDDFS exploits heuristic EPH and needs 5 bits of internal memory for every state, otherwise it cannot obtain the performance claimed in the literature [4]. This is also confirmed by the experimental results on the instances $Peterson(6)$,P4 and $Szyman.(6)$,P4 in Table 1.

In the following, we further analyse the approaches above briefly so as to understand the main reasons for the efficiency of DAAC over DAC and MAP and IDDFS. DAC generates also the whole state space, but does not exploit MPHF; MAP also does not exploit MPHF though it improves DAC. These two algorithms need much more I/O operations than DAAC to carry out state duplicate checking. This, together with their higher I/O complexity than DDAC, reduces significantly their efficiency. Though IDDFS exploits MPHF, it carries out search one level by one level. Beginning with the first level, whenever generating a new level, it regenerates the sub state space from the first level to the new level, and reconstructs the MPHF corresponding to the sub state space, and researches it. This way makes IDDFS get lower efficiency than DAAC. DAAC avoids the disadvantages above by using an efficient framework, an intersection computation technique and a dynamic path management technique. This improves significantly its performance.

8 Discussion

In this section, we first study the parameter in dynamic path management, and then develop two variants of our approach that may output a set of counterexamples. Finally, we discuss those on-the-fly approaches for directly finding all accepting cycles from the graph.

8.1 The ρ Parameter

The ρ parameter needs to be settled in order to execute DAAC. It influences to some extent the performance of DAAC by controlling the size of state blocks moved into or out of the internal memory.

To investigate the parameter's impact on the performance of DAAC, we run DAAC with different values of the ρ parameter. The experimental environment and selected benchmarks are the same as that of Section 7 except for models $Elevator2(7)$,P5 and $Lifts(7)$,P4, because small models do not need to use DPM technique. The experimental results are reported in Table 3.

From Table 3, we observe that the ρ parameter has an impact on the performance of DAAC, and its optimal value varies slightly for different instances. The optimal value is mainly between 0.94 and 0.96, and thus in practical application, we may simply fix the ρ parameter a constant in this area, say 0.94 as in our experiments, without loss of any major efficiency.

8.2 Variants of DAAC

To deal with those systems with a great many of counterexamples, or with the too large size, we consider two variants

TABLE 3
Run Times with Different Values of the Parameter ρ

benchmark	size	$\rho = 0.90$	$\rho = 0.92$	$\rho = 0.94$	$\rho = 0.96$	$\rho = 0.98$
Models with valid property						
$Elev.2(16)$,P4	173, 916, 122	07:12:31	05:05:38	**04:32:11**	05:12:13	05:21:42
$MSC(5)$,P4	119, 663, 657	03:18:06	02:17:21	**02:13:35**	03:18:29	03:39:47
$Phils(16, 1)$,P3	61, 230, 206	03:28:32	02:40:10	02:15:32	**02:00:21**	02:23:43
$Lamport(5)$,P4	74, 413, 141	03:42:19	**01:50:32**	01:55:52	02:30:16	02:36:41
$Peterson(6)$,P4	4, 187, 461, 216	85:25:37	79:34:13	63:13:21	**61:01:32**	65:36:12
$Szyman.(6)$,P4	6, 521, 314, 436	89:15:25	75:36:31	**70:31:15**	79:13:25	82:20:51
Models with invalid property						
$Bakery(5, 5)$,P3	506, 246, 410	03:35:34	02:16:52	01:57:21	**01:40:23**	02:56:21
$Szyman.(4)$,P2	4, 555, 287	01:24:12	00:47:31	00:29:41	00:32:46	**00:26:23**

TABLE 4
Experimental Results of DAAC with Threshold of the Number of Counterexamples

benchmark	size	1	100	1000	10000
$Bakery(5, 5)$,P3	506, 246, 410	00:20:15	00:20:44	00:26:28	01:16:23
$Szyman.(4)$,P2	4, 555, 287	00:08:31	00:08:50	00:12:56	00:29:45
$Elevator2(7)$,P5	43, 776	00:00:06	00:00:11	00:00:12	00:00:12
$Lifts(7)$,P4	73, 473	00:00:05	00:00:08	00:00:10	00:00:10

TABLE 5
Experimental Results of DACC with Time Threshold

benchmark	size	30 seconds	1 minutes	30 minutes	1 hour
$Bakery(5, 5)$,P3	506, 246, 410	0	0	2015	8256
$Szyman.(4)$,P2	4, 555, 287	0	0	4982	4982
$Elevator2(7)$,P5	43, 776	104	104	104	104
$Lifts(7)$,P4	73, 473	120	120	120	120

of our approach by minor modifications. The first is obtained by setting the threshold of the number of counterexamples in DAAC, called Threshold of the Number of Counterexamples-DAAC (TNC-DAAC); and the second by setting a time threshold, called TT-DDAC (Time Threshold-DAAC). In order to show the two variants' performance, we carry out corresponding experiments. The experimental environment is the same as that of Section 7. As six models have no counterexample we only select the remaining four models with invalid properties of Section 7 as benchmarks.

In TNC-DDAC, the threshold of the number of counterexamples is set in advance. When the number of counterexamples reaches the threshold, TNC-DAAC terminates.

Table 4 shows the time consumptions of TNC-DAAC on every model when thresholds of the number of accepting cycles are 1,100, 1,000, 10,000, respectively. From the table, it is observed that for the case that threshold of the number of accepting cycles is 1, the time consumption of TNC-DAAC is less than DAC on all models. Moreover, TNC-DAAC is still faster than MAP and IDDFS on two instances $Elevator2(7)$,P5 and $Lifts(7)$,P4 with the small number of accepting cycles though TNC-DAAC is a bit slower than MAP and IDDFS on other two models. As for other cases of thresholds of the number of accepting cycles, the experimental results are similar to the above.

TT-DAAC works in an anytime way, which means its running time is bounded by time threshold. Table 5 shows the numbers of counterexamples that TT-DAAC finds within different time thresholds for every instance. The experimental results show TT-DAAC needs different time thresholds to find one or more counterexamples for different instances. Generally, the larger scale a system has, the greater time threshold is needed for finding one or more counterexamples.

As for many other application conditions, such as returning all counterexamples of a given length, we can also develop the corresponding variants of our approach by minor revisions, which may be considered as future work.

8.3 On-the-Fly Approaches for Directly Finding All Accepting Cycles from the Graph

One may adopt an on-the-fly approach to find all accepting cycles. Previous on-the-fly approaches (including MAP and IDDFS) which use external memory device are designed to detect one counterexample. Thus, they exit when finding one counterexample, and do not need to search the whole state space. However, in order to find all accepting cycles, whatever approach we use, it has to search the whole state space eventually. This is because we cannot ensure that there is no counterexample in the remaining part of the state space.

To compare our approach with on-the-fly approaches that directly search for all (or a set of) accepting cycles from the graph, we consider the number of I/O operations which they need in the searching. The main factor affecting the number of I/O operations is the check of states.

For large-scale system, to find all accepting cycles, an on-the-fly approach has to store a large set of visited states on the external memory device and make some check against the state set on disk when a state is visited [7], [32]. Because the serial numbers of states in the state set are not continuous generally, locating a state in the set needs to cost averagely $\log m$ I/O operations, where m is the number of states in the set. Thus, every check also needs to cost $\log m$ I/O operations on average. Moreover, a usual on-the-fly approach has to traverse all possible paths to find all

accepting cycles, and a great many of these paths may be ineffective paths. This results in a great number of unnecessary I/O operations.

Though our approach first generates the whole state space by using the algorithm $enumerateBFS()$, the algorithm needs only to traverse the whole state space once. Its aim is to construct a MPHF and get the accepting state set, which is exploited in latter parts. With MPHF, the check for a state needs only one I/O operation in the process of searching for ASCCs and all accepting cycles. In addition, our framework avoids searching for a great many ineffective paths as shown by the example in Section 4.1. Due to all above, our approach is obviously better than on-the-fly approaches that directly search for all (or a set of) accepting cycles from the graph, in terms of the number of I/O operations.

9 Conclusion and Future Work

In this paper, we proposed an I/O efficient approach for detection of all accepting cycles of a large-scale system, namely, DAAC. It exploits a new framework for finding all accepting cycles of a large-scale system, a new linear algorithm for computing intersection of two state sets, a dynamic path management technique, and a minimal perfect hash function. Because so far there is no I/O efficient algorithm for detection of all counterexamples, we compare DAAC with the state-of-the-art algorithms designed to find one counterexample, such as DAC, MAP and IDDFS. Despite the fact that it computes all counterexamples, DAAC has on the whole more satisfying I/O complexity and practical performance. Moreover, because the found counterexamples can be better used to locate root bugs, our approach has important significance for debugging in large-scale system designs.

As for future work, we plan to extend DAAC to a distributed version in order to further improve our approach's efficiency. Since DAAC proceeds by handling every state in the accepting state set and finds all elementary accepting cycles by dealing with every ASCC, it is not difficult to implement a distributed version.

We will also further investigate how to explain and extract information from all (or a set of) accepting cycles, and how to apply it in debugging of large-scale software and hardware systems.

Acknowledgments

The authors would like to thank the editor, associate editor and anonymous reviewers for their highly valuable comments and suggestions. This work was supported by National Natural Science Foundation of China (Nos. 61370072, 61073033, 61472369 and 61421091) and China National 973 Project 2014CB340301. X. Zhang is the corresponding author.

References

[1] M. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proc. Symp. Logic Comput. Sci.*, 1986, pp. 332–344.

[2] J. Barnat, L. Brim, and P. Simecek, "I/O efficient accepting cycle detection," in *Proc. Int. Conf. Comput. Aided Verification*, 2007, pp. 281–293.

[3] L. Brim, I. Cerna, P. Moravec, and J. Simsa, "Accepting predecessors are better than back edges in distributed LTL model-checking," in *Proc. Conf. Formal Methods in Comput.-Aided Des.*, 2004, pp. 352–366.

[4] S. Edelkamp, P. Sanders, and P. Simecek, "Semi-external LTL model checking," in *Proc. Int. Conf. Comput. Aided Verification*, 2008, pp. 530–542.

[5] J. Barnat, L. Brim, P. Simecek, and M. Weber, "Revisiting resistance speeds up I/O-efficient LTL model checking," in *Proc. 14th Int. Conf. Tools Algorithms. Construction Anal. Syst.*, 2008, pp. 48–62.

[6] S. Edelkamp and S. Jabbar, "Large-scale directed model checking LTL," in *Proc. 13th Int. Conf. Model Checking Softw.*, 2006, pp. 1–18.

[7] U. Stern and D. Dill, "Using magnetic disk instead of main memory in the Murφ verifier," in *Proc. Int. Conf. Comput. Aided Verification*, 1992, pp. 172–183.

[8] R. Korf, "Best-first frontier search with delayed duplicate detection," in *Proc. 19th Nat. Conf. Artif. Intell.*, 2004, pp. 650–657.

[9] R. Korf and P. Schultze, "Large-scale parallel breadth-first search," in *Proc. 20th Nat. Conf. Artif. Intell.*, 2005, pp. 1380–1385.

[10] K. Mehlhorn and U. Meyer, "External-memory breadth-first search with sublinear I/O," in *Proc. 10th Annu. Eur. Symp. Algorithms*, 2002, pp. 723–735.

[11] J. Barnat, L. Brim, and J. Chaloupka, "Parallel breadth-first search LTL model-checking," in *Proc. Conf. Autom. Softw. Eng.*, 2003, pp. 106–115.

[12] R. Korf, "Linear-time disk-based implicit graph search," *J. ACM*, vol. 55, no. 6, pp. 1–40, 2008.

[13] S. Leue and M. T. Befrouei, "Counterexample explanation by anomaly detection," in *Proc. 19th Int. Conf. Model Checking Softw.*, 2012, pp. 24–42.

[14] F. Copty, A. Irron, O. Weissberg, N. Kropp, and G. Kamhi, "Efficient debugging in a formal verification environment," in *Proc. 11th Adv. Res. Working Conf. Correct Hardware Des. Verification Methods*, 2001, pp. 275–292.

[15] H. Aljazzar and S. Leue, "Directed explicit state-space search in the generation of counterexamples for stochastic model checking," *IEEE Trans. Softw. Eng.*, vol. 30, no. 1, pp. 37–60, Jan. 2010.

[16] C. Braunstein, U. Kuhne, A. Sulflow, G. Fey, and R. Drechsler, "Increasing the accuracy of SAT-based debugging," in *Proc. Int. Conf. Des., Autom. Test Eur.*, 2009, pp. 1326–1331.

[17] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 6, pp. 1138–1149, Jun. 2008.

[18] J. C. Tiernan, "An efficient search algorithm to find the elementary circuits of a graph," *Commun. ACM*, no. 13, pp. 722–726, 1970.

[19] R. Tarjan, "Enumeration of the elementary circuits of a directed graph," *SIAM J. Comput.*, vol. 2, pp. 211–216, 1973.

[20] F. A. Ehrenfeucht and L. J. Osterweil, "An algorithm for finding the elementary circuits of a directed graph," University of Colorado, Colorado, Tech. Rep. CU-CS-024-23, 1973.

[21] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM J. Comput.*, vol. 4, pp. 77–84, 1975.

[22] H. Weinblatt, "A new search algorithm for finding the simple cycles of a finite directed graph," *J. Assoc. Comput. Machinery*, vol. 19, no. 1, pp. 43–56, 1972.

[23] T. Yamada and H. Kinoshita, "Finding all the negative cycles in a directed graph," *Discrete Appl. Math.*, vol. 118, pp. 279–291, 2002.

[24] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," *Int. J. Softw. Tools Technol. Trans.*, vol. 5, no. 2, pp. 185–204, 2004.

[25] I. Cerna and R. Pelanek, "Distributed explicit fair cycle detection: set based approach," in *Proc. 10th Int. Conf. Model Checking Softw.*, 2003, pp. 49–73.

[26] J. Abello, A. L. Buchsbaum, and J. R. Westbrook, "A functional approach to external graph algorithms," in *Proc. 6th Annu. Eur. Symp. Algorithms*, 1998, pp. 332–343.

[27] R. Diestel, *Grapth Theory*. 3rd ed. New York, NY, USA: Springer-Verlag, 2005.

[28] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

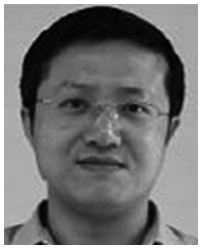
[29] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," in *Proc. 10th Int. Conf. Algorithms Data Structures*, 2007, pp. 139–150.

[30] F. C. Botelho and N. Ziviani, "External perfect hashing for very large key sets," in *Proc. Conf. Inf. Knowl. Manage.*, 2007, pp. 653–662.
[31] S. Jabbar and S. Edelkamp, "Parallel external directed model checking with linear I/O," in *Proc. 7th Int. Conf. Verification, Model Checking, Abstr. Interpretation*, 2006, pp. 237–251.
[32] R. Korf, "Delayed duplicate detection: Extended abstract," in *Proc. Int. Joint Conf. Artif. Intell.*, 2003, pp. 1539–1541.
[33] B. Ding and A. C. Knig, "Fast set intersection in memory," in *Proc. 37th Int. Conf. Very Large Databases*, 2011, pp. 255–266.
[34] R. Pelanek, "BEEM: Benchmarks for explicit model checkers," in *Proc. 14th Int. Conf. Model Checking Softw.*, 2007, pp. 263–267.
[35] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Simecek, "DiVinE—A tool for distributed verification (tool paper)," in *Proc. Int. Conf. Comput. Aided Verification*, 2006, pp. 278–281.
[36] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard template library for XXL data sets," in *Proc. 13th Annu. Eur. Symp. Algorithms*, 2005, pp. 640–651.

Lijun Wu is an associate professor in the University of Electronic Science and Technology of China. His main research interest is formal methods and artificial intelligence.

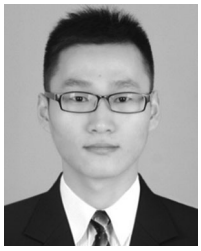
Kaile Su is a professor in Griffith University, Australia. His main research interest is formal methods and artificial intelligence.

Shaowei Cai received the PhD degree in computer science from Peking University, Beijing, China, in 2012. His research interests include optimization, heuristics, and randomized algorithms.



Xiaosong Zhang is a professor and PhD supervisor in the University of Electronic Science and Technology of China. His main research interest is network and information security.

Chenyi Zhang received the PhD degree in computer science from the University of New South Wales, Sydney, Australia in 2009. His research interests are formal methods, computer security, and program analysis.



Shupeng Wang is a graduate student in the University of Electronic Science and Technology of China. His main research interests include formal methods and artificial intelligence.