International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2023, 13(3): 269–296, doi: 10.21655/ijsi.1673-7288.00300

International Journal of Software and Informatics

Research Article

GC-MCR: Directed Graph Constraint-guided Concurrent Bug Detection Method

Shuochuan Li (李硕川)[1], Zan Wang (王赞)[1], Mingxu Ma (马明旭)[1], Xiang Chen (陈翔)[2], Yingquan Zhao (赵英全)[1], Haichi Wang (王海弛)[1], Haoyu Wang (王昊宇)[1]

[1] (College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)
[2] (School of Computer Science and Technology, Nantong University, Nantong 226019, China)
Corresponding author: Zan Wang, wangzan@tju.edu.cn

Abstract Constraint solving has been applied to many domains of program analysis and is further used in concurrent program analysis. Concurrent programs have been widely used with the rapid development of multi-core processors. However, concurrent bugs threaten the security and reliability of concurrent programs, and thus it is of great importance to detect concurrent bugs. The explosion of thread interleaving caused by the uncertainty of the execution of concurrent program threads brings some challenges to the detection of concurrent bugs. Existing concurrent defect detection algorithms reduce the exploration cost in the state space of concurrent programs by reducing invalid thread interleaving. For example, the maximal causal model algorithm transforms the state space exploration problem of concurrent programs into a constraint solving problem. However, it will produce a large number of redundant and conflicting constraints during constraint construction, which greatly prolongs the time of constraint solving, increases the number of constraint solver calls, and reduces the exploration efficiency of concurrent program state space. Thus, this study proposes a directed graph constraint-guided maximal causality reduction method, called GC-MCR. This method aims to improve the speed of constraint solving and the efficiency of the state space exploration of concurrent programs by filtering and reducing constraints using directed graphs. The experimental results show that the GC-MCR method can effectively optimize the expression of constraints, so as to improve the solving speed of the constraint solver and reduce the number of solver calls. Compared with the existing J-MCR method, GC-MCR can significantly improve the detection efficiency of concurrent program bugs without reducing the detection ability of concurrent bugs, and the test time on 38 groups of concurrent test programs widely used by existing research methods can be reduced by 34.01% on average.

Keywords concurrent program; maximal causality reduction; constraint solving; directed graph; conflict constraint filtering

Citation Li SC, Wang Z, Ma MX, Chen X, Zhao YQ, Wang HC, Wang HY. GC-MCR: Directed graph constraint-guided concurrent bug detection method, *International Journal of Software and Informatics*, 2023, 13(3): 269–296. http://www.ijsi.org/1673-7288/300.htm

This is the English version of the Chinese article “GC-MCR: 有向图约束指导的并发缺陷检测方法. 软件学报, 2023, 34(8): 3485–3506. DOI: 10.13328/j.cnki.jos.006865”
Funding items: National Natural Science Foundation of China (61872263); Intelligent Manufacturing Special Fund of Tianjin (20201180)
Received 2022-09-04; Revised 2022-10-13; Accepted 2022-12-14; IJSI published online 2023-09-27

At present, constraint solving, as an important part of symbolic execution, has widely been used in software verification and program analysis, with especially prominent contributions made in analyzing concurrent programs. Featuring fast computing speed and high resource utilization, concurrent programs are increasingly commonly used in the current hard-concurrency era with a wide application of the multi-core architecture. From a micro point of view, thread scheduling of concurrent programs (namely, whether each thread runs at a specific time) is determined by the CPU rather than the users[1]. Therefore, under default program configurations, the execution orders of threads are uncertain to some degree. This makes it difficult for developers to predict program states, easily leaving concurrent bugs inside programs and thus probably causing serious faults. For example, on the day when Facebook was listed in 2012, NASDAQ's trading program suffered from concurrent bugs (the program kept rerunning and entered an endless cycle due to many order withdrawals, resulting in a data race). This caused a 20-minute failure of opening, bringing a huge loss to Facebook. Therefore, in actual project development, it is necessary to fully test concurrent programs so as to avoid possible adverse effects of concurrent bugs. It is of great significance to study automatic bug detection for concurrent software. Relevant research results help detect various concurrent bugs quickly and effectively to ensure the quality and reliability of concurrent software.

In recent years, testing of concurrent programs has attracted wide attention in software engineering, and relevant technology for concurrent bug detection has gradually become a research hotspot. At present, many review papers related to concurrent bug detection have been published in China and abroad. Existing research has deeply explored concurrent bug detection from different angles and made rich achievements. On this basis, some mature tools for concurrent bug detection have also been developed. Currently, common concurrent bugs can be classified into four types: deadlock[4], data race[5], atomicity violation[6], and order violation[7]. In order to detect these concurrent bugs, researchers proposed three kinds of methods: dynamic analysis[8], static analysis[9], and dynamic-static combined analysis[10]. Specifically, the dynamic analysis aims to trigger program exceptions through the dynamic execution of the tested programs, and the triggered exceptions are real. Static analysis is based on relevant pattern rules to extract and compare bug patterns in code to detect bugs[11]. As the main way of static analysis, symbolic execution explores the state space of programs by solving symbolic program path constraints. Dynamic-static combined analysis fully uses the advantages of both dynamic and static analysis. In other words, dynamic analysis can improve the reliability of static analysis, while static analysis can reduce verification loads of dynamic analysis[2, 3].

The main challenge for testing and verification of concurrent programs is the thread interleaving space explosion. It means that the number of possible threads interleaving increases exponentially with the increase in the number of threads and execution time. This makes it difficult to detect concurrent bugs. The key to solving thread interleaving space explosion is to identify redundant thread interleaving[12]. In order to solve the above problem, researchers proposed Partial Order Reduction (POR)[13–15]. This method reduces the size of the overall state space by checking the independence of each state and the behavior of a program. First, POR classifies all thread interleaving into different Mazurkiewicz traces[16], namely different equivalence classes of thread interleaving. Then, POR selects one thread interleaving from each Mazurkiewicz trace for exploration. Mazurkiewicz traces are based on happens-before relations[17], and such relations depend on the release and acquisition of all locks, as well as on conflicting write and read events. For this reason, POR has limitations in identifying interleaving of redundant threads, and it indicates that most of the thread interleaving that only conform to happens-before relations are redundant[18]. In view of this problem, Huang *et al*. proposed the algorithm of Maximal Causality Reduction (MCR)[18]. In MCR, a new criterion is developed

based on Mazurkiewicz traces: maximal causality relation. Under this criterion, maximally possible equivalent thread interleaving sets in Mazurkiewicz traces are divided according to values of read and write events. MCR converts exploration of state space of concurrent programs into constraint solving and detects concurrent bugs stably and efficiently. However, the extremely large state space of concurrent programs will result in extremely strict and complex constraints imposed by MCR on events. Complex constraint construction will easily produce many redundant or even conflicting constraints, leading to time-consuming constraint solving. According to experimental statistics, the time spent on constraint solving by MCR accounts for about 89.21% of the total running time of MCR. Or even, a lot of time is consumed in solving conflicting constraints but without a solution. This significantly reduces the efficiency of concurrent bug detection.

In view of the above problems, focusing on constraints in MCR, this study improves the efficiency of concurrent bug detection by filtering conflicting constraints and reducing redundant ones and finally designs a method, namely GC-MCR. GC-MCR first constructs a directed graph according to the basic constraints of a program (namely, happens-before constraints, lock mutex constraints, etc.). Then it parses and splits read-write constraints generated by MCR[18]. Specifically, the read-write constraints are used to construct a read-write constraint tree, and the value of each node in this constraint tree is obtained according to the directed graph of the basic constraint relationship of the program. This value indicates whether the subtree with the current node as the root node conflicts with the basic constraints of the program. Finally, whether to filter or reduce constraints can be determined according to the value of each node in the read-write constraint tree. GC-MCR effectively reduces the time of single constraint solving and even the number of calls of constraint solvers by filtering conflicting constraints and reducing expressions of redundant constraints. Thus, it can ensure the capacity of concurrent bug detection and improve relevant detection efficiency.

In this study, 38 concurrent programs widely used in the existing research are selected for testing. Bugs contained in these concurrent programs include deadlock, data race, atomicity violation, and order violation. From the test results, GC-MCR is clearly better than the existing methods not only in the number of calls of constraint solvers and time of individual calls but also in the efficiency of exploring state space of concurrent programs. Compared with J-MCR, in the case of no loss in the total number of times of scheduling, namely, no loss in the capacity of detecting concurrent bugs, GC-MCR can reduce the number of times of constraint solving and total calling time by 34.01% on average. The main contributions of this study are summarized as follows:

- A representation method of concurrent constraints is designed based on a directed graph. In this method, potential unsolvable concurrent constraints are identified from the directed graph to optimize redundant concurrent constraints, so as to improve the accuracy of concurrent-constraint representation.
- A new concurrent-bug detection method called GC-MCR is proposed. GC-MCR constructs complete constraints of concurrent programs based on the directed-graph representation method proposed in this study. Then, it filters and reduces such constraints to effectively reduce the overhead of constraint solving and improve the efficiency of exploring the state space of concurrent programs.
- GC-MCR (https://github.com/WingedVampires/GC-MCR) is implemented and open-sourced and verified on 38 concurrent test programs. According to the test results, compared with the existing research methods, GC-MCR can detect bugs in concurrent programs faster, with an average improvement of 34.01% in performance.

1 Background Knowledge

This study mainly aims to construct complete concurrent program constraints by directed-graph representation and filter and reduce such constraints to decrease the overhead of constraint solving and improve the efficiency of exploring the state space of concurrent programs. Theories and methods related to the proposed method include the Maximal Causal Model (MCM) serving as the basic model of GC-MCR, MCR[18] based on MCM, and the tool Z3[20] for constraint construction and solving.

1.1 MCM

MCM[5, 19] constructs the largest and reasonable causal model according to a given program execution trace. It contains all the traces generated by all the concurrent programs that can generate the original trace. In other words, if a concurrent program P can generate the original trace, then all the traces generated by P are included in the MCM. Therefore, MCM can find thread interleavings that have not yet been executed in the large state space. In MCM, an event is an atomic operation performed by a thread on a concurrent object. Its attributes include the thread that accesses a concurrent object (*Thread*), the type of operation executed on the concurrent object (*OP*), the concurrent object accessed by the event (*Target*), and the operation value executed on the concurrent object (*Data*). Two events s_1 and s_2 are said to be equal if and only if all the attributes corresponding to them are equal. Types and corresponding descriptions of events in MCM are shown below:

- *begin*(t): the first event in thread t;
- *end*(t): the last event in thread t;
- *read*(t, x, v): the value of x read by the thread t is v;
- *write*(t, x, v): the value that the thread t writes to the variable x is v;
- *lock*(t, l): the thread t acquires the lock l;
- *unlock*(t, l): the thread t releases the lock l;
- *fork*(t, t'): the thread t creates a new thread t'; and
- *join*(t, t'): the thread t is blocked until t' ends its execution.

In MCM, an execution trace of a program can be abstracted as a sequence of events. It is also stipulated that an execution trace needs to satisfy order consistency, namely, read-write consistency, lock mutex, and happens-before principle. Specifically, read-write consistency needs to ensure that the value of a read event (read) is that of the write event (write) preceding the read event in the execution trace, being closest to the read event, and accessing the same shared memory address as the read event. Lock mutex needs to ensure that there is always a lock-acquisition event (acquire) of the same thread and the same lock variable preceding each lock-release event (release) to match this release event, and that each acquire-release pair cannot be interleaved with other lock pairs that access the same lock variable. The happens-before principle needs to ensure that a program is executed according to the code order, that the write operation of a volatile-modified variable happens before the read operation of the variable, and that the unlocking operation of a lock happens before the locking operation of the same lock. In addition, if operation A happens before operation B, and operation B happens before operation C, then A should happen before C. All operations before calling B.*start*() in the thread A happen before all operations in the thread B. The thread B.*join*() will execute the thread B first, indicating that all operations in the thread B happen before those after the *join* operation in the thread A.

In MCM, *feasible*(τ) is the smallest set of all program-execution-generated order-consistent traces that can produce the execution trace τ, and it is the final result of MCM. *feasible*(τ) needs to satisfy prefix closedness and local determinism. Prefix closedness assumes that an

event is independent, indivisible, and generated sequentially based on program execution. Thus, *feasible*(τ) must be prefix-closed. Local determinism assumes that the execution of a concurrent operation is determined by previous events in the same thread and can happen at any time after them.

1.2 MCR

By constructing maximal causal models of concurrent programs, MCR transforms the exploration of the state space of programs into constraint solving. Figure 1 illustrates this algorithm. With the input of the program to be tested, MCR obtains the execution trace of the program according to the specified thread interleaving through the model checker. The scheduler in the model checker specifies a thread interleaving by controlling the scheduling of threads. Then a new seed thread interleaving is generated and added to the state space of the program under test through the maximal causality engine so that this interleaving can be reused by the model checker, and a new seed interleaving can be obtained by running. For a thread interleaving s, through MCM introduced earlier, a unique and largest feasible interleaving set can be obtained, which is denoted as *MaxCausal*(s). The new seed interleaving generated by each MCR running is a feasible interleaving in *MaxCausal*(s).

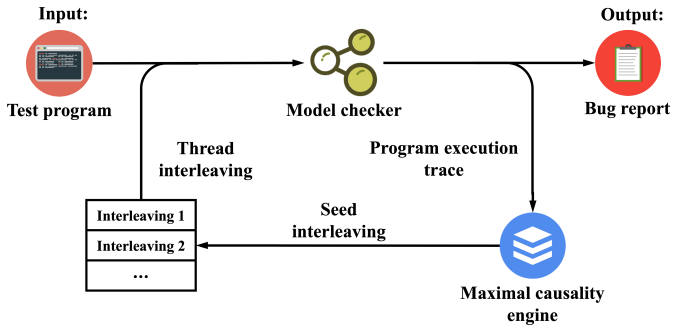


Figure 1 Execution process of MCR

During the running of a concurrent program, interleaved execution of different threads will produce a series of thread interleavings. Each triggered thread interleaving will make the program reach a specific state. Here, it is stipulated that when a new interleaving is generated before the execution of a program, if the program enters a new state by reading a different value from the same variable, then the algorithm needs to add this state to the state space of the tested program. When the state space of the tested program is full, indicating that all the seed interleavings have been explored with no new seed interleaving being generated, it can be concluded that the state space of the tested program has completely been covered. Here, MCR proposes a new constraint, namely Race. Specifically, it is assumed that *COP*(x, y) represents a read-write pair of a global variable in different threads. Then for every existing global variable of *COP* relationship in the program, each read event R of such a variable needs to be constrained. In other words, all read events happening before R should read the same value as before, and R should read a completely different value. As a program may reach a different state every time reading a new value, a state tree of the program can be obtained by constantly reading new values during constraint solving of the program. This tree can cover all possible states in the program.

Reasons for the large state space of a concurrent program can be divided into two aspects. Firstly, combinations of threads are diverse, most of which are feasible, and the scheduling of threads is uncertain. Secondly, many equivalent thread interleavings exist in the state space of a

concurrent program. Except for different scheduling orders of threads, data read are the same, and it means that these equivalent threads cannot make the program reach a new different state. Many equivalent thread interleavings will prolong the execution of other algorithms. MCR will also execute a tested program repeatedly and generate new seed interleaving by intervening in thread scheduling, executing all schedules, and checking the feasibility of the current schedule. However, this algorithm recognizes that the program has reached a new state only when it reads a new value and then adds the current thread schedule to the state space of the current program, so as to minimize the state space.

1.3 Construction and solving of constraints

In this study, constraints in MCR are divided into basic constraints of programs and read-write constraints. Basic constraints of programs are an embodiment of constraints on happens-before principles in the programs, including program order constraints, thread constraints, and lock constraints. Read-write constraints are to restrict different threads to reading and writing the same shared variables according to a specified read-write requirement. Specifically, the read-write requirement is that only one read event R among the read events of all threads in a program reads the new value, while read events that happen before R need to read old values. As shown in the program in Figure 2, T1 and T2 are two threads. The basic constraints of the program in this figure are as follows: $x1 < x2$, $x3 < x4$, $x4 < x5$, and $x5 < x6$, or in other words, the program is executed according to the order of the code in the same thread. Suppose that x in Line 4 should read the value 1. Then, the read-write constraints are as follows: $x3 < x1$ and $x1 < x4$, indicating that assigning 1 to x needs to be executed before Line 4 and after assigning 0 to x in Line 3. Basic constraints and read-write constraints are in and relations, namely, juxtaposition relations. Only scheduling sequences solved by both the two kinds of constraints are executable.

```
T1          T2
1: x = 1    3: x = 0
2: y = 2    4: if (x > 0) {
            5:     y − −
            6:     x = 7
               }
```

Figure 2 Example program for constraint construction

In this study, Z3[20] is used to solve constructed constraints. Z3 is an open-source constraint solver developed by Microsoft, and it is a general solver used to solve deterministic problems of first-order logic combinations. Given partial constraints, Z3 can find a set of solutions that satisfy these conditions. Using Z3 for problem solving requires writing a Z3 script. The script contains a specific command sequence, which will be parsed by the solver using a specific algorithm. Finally, the solver will return the solution to the formula proposition and the assignment from the theoretical solver of Z3. New constraints obtained by GC-MCR are solved by using Z3, and then new scheduling sequences are obtained according to relevant solving results, so as to explore the state space of concurrent programs.

2 GC-MCR

2.1 Research motivation

In order to fully explore the state space of a concurrent program, by analyzing existing execution traces, MCR[18] generates constraints that satisfy MCM and Race according to constraint relations between different events and then solves them with a constraint solver. Finally, the concurrent program is further explored according to solving results. Specifically, if a constraint

is solvable, program scheduling is controlled according to the solved thread interleaving; otherwise, thread interleaving fails to happen, and the algorithm will skip over the current constraint to solve other constraints. MCR can detect potential concurrent bugs in concurrent programs through a small number of times of scheduling. However, from the existing research work[21] and the reproduction and in-depth analysis of MCR in this work, MCR has coarse granularity in constructing program constraints, producing many redundant or unsolvable constraints. This results in an excessive number of times of constraint solving, consuming many additional computational resources. If unsolvable constraints are filtered, or redundant ones are reduced before solving, a lot of computational resources will be saved. Therefore, much solving overhead can be saved by filtering unsolvable constraints or reducing redundant ones before solving, so as to improve the efficiency of the detection of concurrent bugs. For this reason, this study carries out test verification on three test programs with different state spaces: hashcode, numberaxis, and dekker. The test results show that MCR solved constraints 32, 242, and 492 times in the above test programs, respectively. However, according to further analysis of these constraints, 21, 67, and 150 constraints respectively in these test programs can be filtered by constraint analysis without the necessity of being solved with the constraint solver, and other constraints can be reduced to varying degrees. Based on the above findings, by filtering and reducing the above constraints, this study finally reduces program execution time by 51.78%, 28.29%, and 58.35% respectively, with a significant improvement in efficiency of exploring state space of concurrent programs. This study also carries out tests on other programs and ultimately draws the same conclusion. In order to illustrate the basic principles of the proposed method, the study uses the program dekker as an example for subsequent method introduction.

From the tests, some of the constraints generated by MCR have obvious conflicts, and these conflicts can be handled in a simple and effective way. For example, Figures 3 and 4 illustrate one basic constraint of the program and two read-write constraints in one constraint solving of dekker. Each assert statement is composed of the keyword assert and a sub-constraint. Each sub-constraint is represented by a prefix expression. In the expression, the content in the innermost set of parentheses is a constraint unit, and each constraint unit can be combined by using and/or to form a sub-constraint. For example, in Figure 4(b), (or ($> x1\ x9$) ($< x3\ x6$)) is a sub-constraint, where ($> x1\ x9$) is a constraint unit, actually meaning that event $x1$ happens after event $x9$. In Figure 3, ($> x1\ x2$) is both a constraint unit and a sub-constraint. There is an and-relation between assert statements, namely that the overall constraint is satisfied if and only if each constraint is true.

(assert ($> x1\ x2$)) (assert ($< x5\ x6$)) (assert ($< x7\ x8$))
(assert ($> x3\ x5$)) (assert ($> x6\ x8$)) (assert ($< x4\ x6$))
(assert ($> x3\ x7$)) (assert ($< x6\ x2$)) (assert ($> x9\ x4$))
(assert ($< x4\ x2$)) (assert ($> x3\ x6$))

Figure 3 Example of basic constraint of program

(assert (and (or (or ($< x8\ x7$) ($< x9\ x4$))(and ($> x1\ x8$)
($> x3\ x5$)))(and (or ($> x1\ x4$) ($> x1\ x6$))($< x3\ x6$))))
(a) Conflicting constraint

(assert (or ($> x1\ x9$) ($< x3\ x6$))
(b) Redundant constraint

Figure 4 Example of read-write constraints

Conflicting constraints. $x8 < x7$, $x4 > x9$, and $x3 < x6$ in the read-write constraint in Figure 4(a) conflict with $x8 > x7$, $x9 > x4$, and $x3 > x6$ in the basic constraint of the

program in Figure 3, respectively, while other constraints are consistent with the basic constraint of the program. By logical operation, the final result is false, indicating conflicts with the basic constraint of the program. In other words, read-write constraints do not hold under the condition of satisfying the basic constraint of the program. Thus, such read-write constraints conflicting with the basic constraint of the program are defined as conflicting constraints. Conflicting constraints are unsolvable. The concurrent test program dekker contains 150 conflicting constraints. Identifying and filtering these conflicts before calling the constraint solver can save a lot of computational resources.

Redundant constraints. From further analysis, $x3 < x6$ in the read-write constraint in Figure 4(b) conflicts with $x3 > x6$ in the basic constraint of the program, while $x1 > x9$ does not conflict with the basic constraint of the program. Moreover, the read-write constraint in Figure 4(b) holds under the condition that the basic constraint of the program is satisfied. Therefore, such read-write constraints not conflicting with basic constraints on the whole but containing partial sub-constraints that conflict with basic constraints are defined as redundant constraints. Redundant constraints can be reduced according to the basic constraints of programs. The redundant constraint in Figure 4 can be reduced to $(> x1\ x9)$. For constraints after the reduction, the constraint solver can solve them faster, saving the computational resources of the program in constraint solving. Moreover, the time overhead of filtering and reducing constraints is much less than that of solving original constraints with the solver.

Based on the above analysis, this study proposes a method called GC-MCR. At first, constraints are divided into read-write constraints and basic constraints of programs. Then, the basic constraints are constructed into directed graphs before the read-write constraints are built into prefix trees. Finally, whether conflicting and redundant constraints exist in the read-write constraints is judged according to the directed graphs. If there is a conflicting constraint, the current constraint is filtered out. If there is a redundant constraint, the redundant constraint is reduced and then the constraint solver is called again for solving, so as to improve the efficiency of concurrent bug detection.

2.2 Description of the method

2.2.1 *Overall framework: Introduction to input, output, and overall process*

The input of GC-MCR is a concurrent test program, and its output is a concurrent bug report of the program. First of all, the model checker executes the concurrent test program to obtain an initial execution trace of the program. The maximal causality engine analyzes and processes the execution trace to generate constraints that satisfy MCM and Race constraints. Then, GC-MCR classifies the constraints into read-write constraints and basic constraints of the program. The basic constraints of the program are analyzed and reconstructed into a directed graph. Read-write constraints are analyzed, split, and reconstructed into a read-write constraint tree. After that, the constraint filter calculates the value of each node in the read-write constraint tree according to the directed graph and determines whether to skip over the current solving or reduce the current constraint according to the value of the root node of the read-write constraint tree. If constraint solving is carried out, the obtained thread interleaving will be stored in the thread-interleaving set. Finally, by executing thread interleavings in this set successively, the model checker acquires program execution traces and detects concurrent bugs in the program. If a concurrent bug is detected, a bug report is output; otherwise, the above operations are repeated. Figure 5 illustrates the specific workflow of GC-MCR.

2.2.2 *Reconstruction of basic constraints based on directed graph*

The constraints in Figure 3 are basic constraints of the program generated by GC-MCR, and all the basic constraints satisfy the happens-before principle. According to the definition of

an order-consistent model, constraints in the program must follow the happens-before principle. For this reason, there should be no constraints that violate the basic constraints of the program. Figure 6 is a directed graph generated by this method, illustrating the sequential relationship of various scheduling points in the program under the happens-before principle.

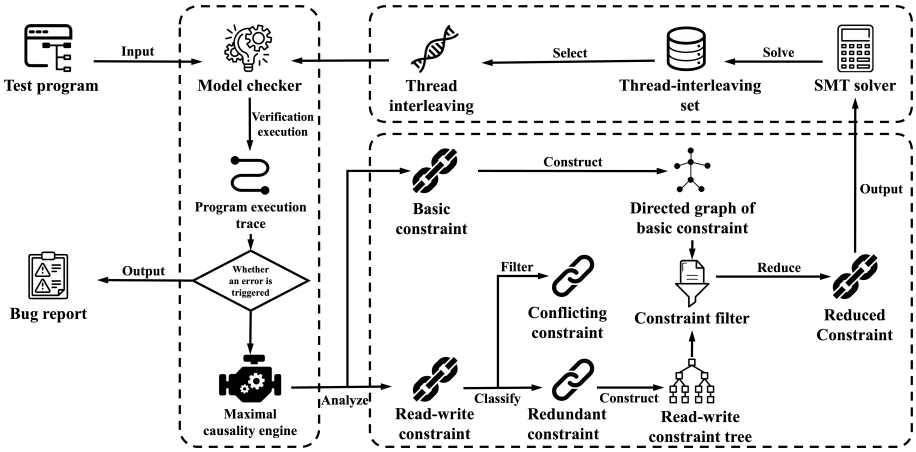


Figure 5 Flow chart of concurrent bug detection based on directed-graph constraints

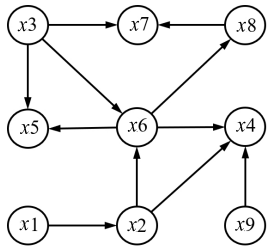


Figure 6 Directed graph of basic constraints

Order inequalities in basic constraints of a program satisfy transitivity; however not all pairs of scheduling points occur in such order relations. Therefore, it is difficult to conclude the order relations of all scheduling points through size representation on the line of natural numbers. In order to clearly show the order relations of scheduling points, the proposed method transforms these order relations into directed graphs for representation. In terms of implementation, a directed graph is built by using an adjacency list, and a new node class is defined, which contains the scheduling point corresponding to the current node and the set of nodes pointing directly to this scheduling point in the directed graph. In Figure 3, if the constraint $x1 > x2$ exists, the edge of $x1$ pointing to $x2$ is constructed in the directed graph. The process is repeated until the directed graph is finally constructed. As shown in Algorithm 1, the relation between each pair of scheduling points in the basic constraints of the program is first analyzed. Then, the corresponding nodes are found in the graph. Finally, points-to relations between nodes are stored in the adjacency list. Through the above steps, relations of basic constraints of the program can be represented by a directed graph. On this basis, just by traversing the directed graph, this study can obtain relations between thread scheduling points in the program, which will be used for subsequent processing of read-write constraints.

Algorithm 1. Reconstruction of basic constraints based on directed graph.

Input: Φ_{base}: Basic constraints of program;
Output: G: Directed graph composed of adjacency list.
1. $G \leftarrow \varnothing$;
2. **for** φ_i in Φ_{base} **do**
3. *option, target, next* $\leftarrow$ *analysisConstraint*(φ_i);
4. *targetNode, nextNode* $\leftarrow$ *findNode*(*target, next, G*);
5. *targetNode.largeNodes.add*(*nextNode*);
6. **end for**
7. **return** G

2.2.3 *Analysis and splitting of read-write constraints*

The generation of read-write constraints is the core step of GC-MCR. This step ensures that each read event R of all shared variables meets the two conditions: the read event R reads a value that has not been read before, and all read events that happen before R read the same values as before. This ensures that only one read event reads the new value, while the other read events read the original values. For example, for a variable x, if a new value needs to be read, the event of writing a new value to x should be arranged before the event of reading x. If the original value needs to be read, the event of writing a new value to x should be arranged after the read event. Therefore, GC-MCR can change the order of scheduling points by controlling read and write events.

However, there are many restrictions and possible cases to make a new value read in a read event, including both constraints to guarantee a new value written to x and those to keep values of other read events unchanged. Thus, read-write constraints are quite complex. As shown in Figure 4(a), the read-write constraint is an `or-and` combined constraint. Under the existing methods, such read-write constraints and previously generated basic constraints are introduced into the constraint solver for solving. After order relations of scheduling points are obtained, they are parsed into new thread scheduling sequences for directed thread scheduling. Finally, the whole state space of programs can be explored. However, the existing methods ignore the relationship between read-write constraints and basic constraints of programs, with obviously conflicting constraints and redundant ones being included. Conflicting constraints require no calling of the constraint solver for solving, and redundant ones can be solved after reduction. Filtering conflict constraints and reducing redundant ones can effectively lessen the time of single solving and even the number of calls of the constraint solver, so as to obviously improve the efficiency of concurrent-bug detection.

Therefore, GC-MCR parses read-write constraints. Figure 7 illustrates a prefix tree constructed according to the read-write constraint in Figure 4(a), where non-leaf nodes are the

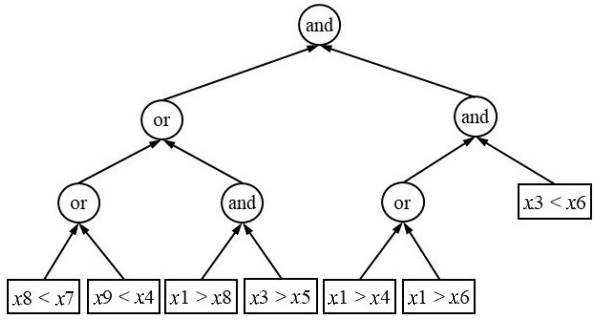


Figure 7 Read-write constraint tree

logical operators and/or, and leaf ones are order relations of scheduling points. It is then used to traverse the directed graph of basic constraints of the program for conflict analysis. Details can be found in Algorithm 2. For example, suppose that, in one of the leaf nodes, there is $x_3 > x_9$. Check whether there is $x_9 > x_3$ in Figure 6, or whether there is a path from x_9 to x_3 (Line 6). If so, set this leaf node to false, indicating a conflict in the subexpression. If not, set this leaf node to true, indicating no conflict in the subexpression. This operation is executed at each leaf node, replacing the results of all leaf nodes with true/false. Meanwhile, each leaf node should also store the constraint expression of the current node (Lines 7–8), for subsequent reduction of constraint expressions. After conflict analysis, the values of all the leaf nodes should become true/false. In this case, the whole tree is an operation tree composed of and/or and true/false. Finally, all the study needs to do is to operate with the operation tree. If the result is false, the read-write constraint conflicts with basic constraints of the program, and thus the study directly jumps over the calling of the constraint solver this time. Otherwise, the current constraint is reduced. Constraints that are filtered out need no calling of the constraint solver, and this part of the time is optimized by the filtering algorithm to improve the efficiency of concurrent bug detection.

Algorithm 2. Analysis and splitting of read-write constraints.

Input: φ_{rw}: A read-write constraint of the program; G: a directed digraph of basic constraints of the program;

Output: T: A read-write constraint tree represented by prefix expressions, in which each node contains the Boolean value of the current node and the constraint from the leaf node to the current node.

1. $T \leftarrow \varnothing$;
2. *list* $\leftarrow$ *split*(φ_{rw});
3. **for** s in *list* **do**
4. **if** $s \in \{<, >\}$ **then**
5. *target, next* $\leftarrow$ *analysisConstraint*(s);
6. *isConflict* $\leftarrow$ *isConflictWithOrderMap*(*target, next, G*);
7. *expression* $\leftarrow$ *getConstraintExpression*(*s, target, next, isConflict*);
8. *T.add*(*Pair*(*isConflict, expression*));
9. **else**
10. *T.add*(*Pair*(s, s));
11. **end if**
12. **end for**
13. **return** T;

2.2.4 *Reduction of constraint expressions*

For the reduction of redundant constraints, GC-MCR will first reduce conflicting subexpressions and then call the constraint solver to solve the reduced constraint expressions and the expressions of the basic constraints of the program.

In this study, each node of the read-write constraint tree contains not only the value indicating whether a subtree with the current node as the root node satisfies the basic constraints of the program but also the constraint expression of the subtree. The constraint expression in a leaf node of the constraint-expression prefix tree is called a subexpression. First, values of leaf nodes in which subexpressions conflict with the expressions of the basic constraints of the program are set to false, while those of the other leaf nodes are set to true. As shown in Figure 8, the values of leaf nodes of the prefix tree after assignment are true/false. In the figure, T represents true, and F represents false. Then, the following five cases are considered based on the operator of a parent node and the values of its two child nodes.

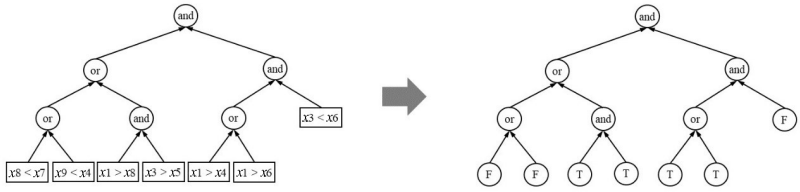


Figure 8 Example of prefix-tree replacement

(1) The operator of a parent node is `and`, and values of both child nodes are `true`. As shown in Figure 9(a), the value of the parent node is set to `true`, and the constraint expression of the parent node is set to that of the two child nodes of an `and`-relation.
(2) The operator of a parent node is `or`, and values of both child nodes are `true`. As shown in Figure 9(b), the value of the parent node is set to `true`, and the constraint expression of the parent node is set to that of the two child nodes of an `or`-relation.
(3) The operator of a parent node is `or`, and values of both child nodes are `false`. As shown in Figure 9(c), the value of the parent node is set to `false`, and the constraint expression of the parent node is set to null.
(4) The operator of a parent node is `and`, and at least one of the two child nodes has the value of `false`. As shown in Figure 9(d), the value of the parent node is set to `false`, and the constraint expression of the parent node is set to null.
(5) The operator of a parent node is `or`, and only one of the two child nodes has the value of `true`. As shown in Figure 9(e), the value of the parent node is set to `true`, and the constraint expression of the parent node is set to that of the child node with the value of `true`.

The above operation is repeated until only the root node exists in the read-write constraint tree. Then, whether to call the constraint solver is judged according to the value of the root node. If the value is `false`, it is not necessary to call the solver. Otherwise, the constraint solver is used for solving based on the constraint expression of the root node, namely, the simplest read-write constraint expression, together with the expressions of the basic constraints of the program. In the present example, the value of the final root node is `false`. Thus, the study can skip the constraint-solving step directly to execute subsequent operations of GC-MCR, so as to detect concurrent bugs in the concurrent program.

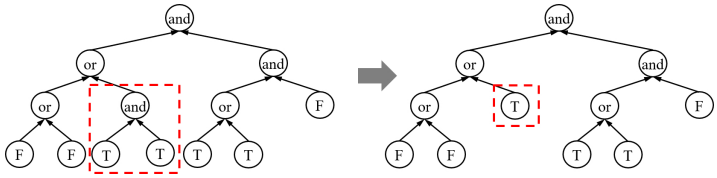
3 Test Design and Result Analysis

3.1 Research questions

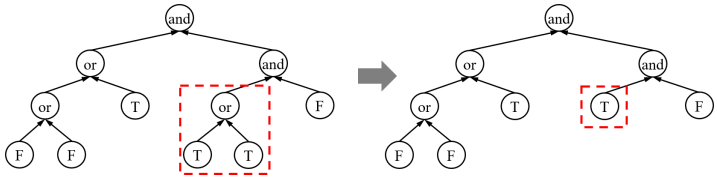
In order to verify the effectiveness of GC-MCR, this study designs three research questions in terms of the number of times of constraint solving, running time of programs, and number of times of program scheduling.

RQ1: Is GC-MCR effective in reducing the number of times of constraint solving and computational resource consumption of the constraint solver?

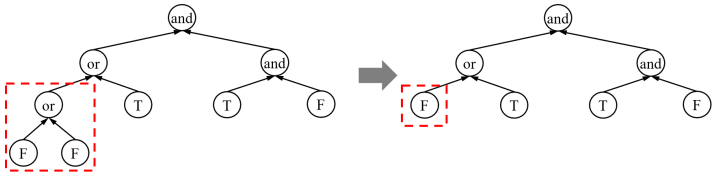
Calling constraint solvers for constraint solving is more time-consuming than program scheduling. For this reason, reducing the number of times of constraint solving to save computational resources of the constraint solver is of great significance for concurrent bug detection. Failing to process constraints before solving, the existing method[18] solves many conflicting constraints, which can be filtered out in advance, by the constraint solver. This increases the number of times of constraint solving, thus reducing the efficiency of exploring the state space of concurrent programs. Therefore, this study hopes to analyze whether GC-MCR can effectively



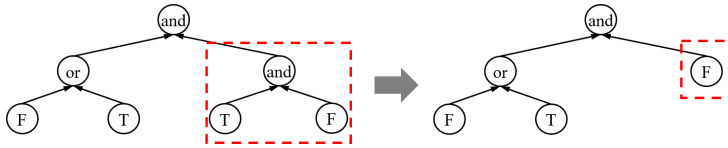
(a) The operator of a parent node is `and`, and values of both child nodes are `true`



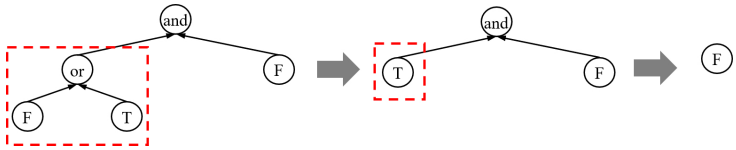
(b) The operator of a parent node is `or`, and values of both child nodes are `true`



(c) The operator of a parent node is `or`, and values of both child nodes are `false`



(d) The operator of a parent node is `and`, and at least one of the two child nodes has the value of `false`



(e) The operator of a parent node is `or`, and only one of the two child nodes has the value of `true`

Figure 9 Example of prefix-tree simplification

reduce the number of times of calling the constraint solver and save computational resources of the solver, compared with the existing methods.

RQ2: Can GC-MCR effectively reduce the time of concurrent bug detection and improve the efficiency of exploring the state space of concurrent programs?

The time required by a program to detect concurrent bugs is an important index to evaluate the efficiency of concurrent bug detection. Through the statistics of the number of times of constraint solving and running time of the program, the study can also analyze the relationship between the two, which means whether time consumption of concurrent-bug detection can be reduced by reducing the number of times of constraint solving, as well as factors that affect concurrent bug detection. Therefore, this study hopes to analyze from multiple angles whether

GC-MCR can effectively reduce the time of concurrent-bug detection to improve the efficiency of exploring the state space of concurrent programs, compared with the existing methods.

RQ3: Can GC-MCR guarantee the concurrent bug detection capacity of the existing method?

The existing methods have obtained certain effects in concurrent bug detection, indicating that they can accurately detect bugs in concurrent programs by the least number of times of scheduling. The number of times of scheduling of concurrent programs is also a major index to measure the capacity of concurrent bug detection. If compared with the existing method, GC-MCR can detect the same bugs with the same or even smaller number of times of scheduling, the capacity of concurrent bug detection does not decline. Therefore, this study hopes to verify whether GC-MCR can have the same or even higher capacity of concurrent bug detection, compared with the existing method.

3.2 Test configuration

Test objects. In this study, tests are carried out on 38 test programs with concurrent bugs, which are collected from the concurrency dataset widely used in the existing research[18, 22, 23]. Table 1 lists the details of these classes. Specifically,

- The "ID" column represents the serial number of test programs;
- The "test program" column represents the name of the test program;
- The "number of lines of code" represents the number of lines of code in the test program (counted by the tool Statistic);
- The "number of threads" column represents the number of threads in the test program;
- The "number of variables" column represents the number of variables defined in the test program, including variables from their parent classes. It should be noted that the statistics of variables do not distinguish between the access rights of these variables (for example, public or private). This is because all instance variables are shared correctly;
- The "number of methods" column represents the number of public methods in the test program, including those from parent classes; and
- The "concurrent bug types" column represents the type of the concurrent bug in the test program.

In particular, in order to evaluate the effectiveness and scalability of GC-MCR, the study further verifies this method on the real-world program Weblech. Weblech is a website download tool, including a test driver that starts the server, executes client requests, and terminates programs. Compared with other test programs, Weblech has more program events and a larger state space for exploration.

Contrast method. The contrast method is the Java-based implementation of MCR, converting the exploration of the state space of concurrent programs into constraint solving to detect concurrent bugs. GC-MCR proposed in this study not only converts the exploration of the state space of concurrent programs into constraint solving but also proposes a directed graph-based representation method. With this representation method, GC-MCR constructs complete constraints of concurrent programs and then filters and reduces them to decrease the time overhead of constraint solving.

All the related tests in this study are carried out in a machine running Windows 10 Enterprise Edition (64 bit) with a quadcore Intel® Core™ CPU i7-7500U @ 2.70 GHz 2.90 GHz and 8 GB of memory. The timeout for each program execution is set to 90 minutes. In order to minimize the influence of random factors, each test is repeated 10 times, and the average value is taken as the final result.

Table 1 Description of GC-MCR test object set

ID	Test program	Number of lines of code	Number of threads	Number of variables	Number of methods	Concurrent bug types
1	account	377	10	1	19	Atomicity
2	airline	182	5	4	15	DataRace
3	alarmclock	372	9	19	21	DataRace
4	allocation	347	2	2	14	Atomicity
5	array	104	2	20	2	DataRace
6	bakery	126	2	6	5	Atomicity
7	boundedbuffer	159	3	1	2	Deadlock
8	bubblesort	160	4	4	7	DataRace
9	checkfield	65	2	2	4	DataRace
10	clean	145	12	10	9	DataRace
11	consistency	59	3	2	2	Atomicity
12	counter	57	2	2	2	DataRace
13	critical	117	2	1	2	Atomicity
14	cyclic	72	4	1	3	DataRace
15	datarace	16	2	10	2	DataRace
16	deadlock	64	2	3	4	Deadlock
17	dekker	118	2	6	2	Atomicity
18	dirkaccount	167	2	7	10	Deadlock
19	fileappender	410	2	7	33	Atomicity
20	hashcode	2,461	2	1	3	Atomicity
21	lamport	160	2	9	2	Atomicity
22	mergesort	384	3	11	13	DataRace
23	numberaxis	1,637	2	43	110	Atomicity
24	peruserpooldatasource	682	2	35	65	DataRace
25	peterson	85	2	5	2	Atomicity
26	pingpong	294	5	9	13	DataRace
27	pool	243	2	4	2	DataRace
28	rax	95	2	2	2	Deadlock
29	readerswriters	608	3	12	18	DataRace
30	replicatedcasestudies	1,427	4	1	2	Deadlock
31	rvexample	75	2	3	2	DataRace
32	sharedobject	67	2	1	2	DataRace
33	sharedpooldatasource	516	2	30	51	Atomicity
34	simple	65	2	2	2	Atomicity
35	store	79	2	1	4	Atomicity
36	transfer	91	2	3	7	Deadlock
37	waitnotify	99	4	1	1	DataRace
38	Weblech	35K	3	49	90	DataRace

3.3 Result analysis

RQ1: Is GC-MCR effective in reducing the number of times of constraint solving and computational resource consumption of the constraint solver?

In response to RQ1, this study systematically applies GC-MCR to each test program and calculates the number of times of constraint solving and corresponding constraint-solving time needed to detect the first concurrent bug, thus checking the difference between the two methods. This study compares GC-MCR with J-MC. Test results are shown in Tables 2 and 3. In the tables, the number of times of GC-MCR constraint solving and the number of times of J-MCR constraint solving represent the number of times of constraint solving required to detect the first concurrent bug in a test program by using GC-MCR and J-MCR, respectively. The reduced number of times of constraint solving represents the number of times of constraint solving that can be reduced under GC-MCR, compared with that under J-MCR. The percentage of reduced constraint solving represents the percentage of the reduced number of times of constraint solving

Table 2 Comparison of number of times of constraint solving between GC-MCR and J-MCR

ID	Test program	Number of times of GC-MCR constraint solving	Number of times of J-MCR constraint solving	Reduced number of times of constraint solving		Percentage of reduced constraint solving (%)	
1	account	12	12	–	0	–	0.00
2	airline	2,303	2,530	↓	227	↑	8.97
3	alarmclock	24,526	26,799	↓	2,273	↑	8.48
4	allocation	30	36	↓	6	↑	16.67
5	array	399	437	↓	38	↑	8.70
6	bakery	75	93	↓	18	↑	19.35
7	boundedbuffer	16	16	–	0	–	0.00
8	bubblesort	272	286	↓	14	↑	4.90
9	checkfield	4	14	↓	10	↑	71.43
10	clean	25	25	–	0	–	0.00
11	consistency	8	12	↓	4	↑	33.33
12	counter	66	67	↓	1	↑	1.49
13	critical	2	3	↓	1	↑	33.33
14	cyclic	100	100	–	0	–	0.00
15	datarace	5	15	↓	10	↑	66.67
16	deadlock	1	1	–	0	–	0.00
17	dekker	342	492	↓	150	↑	30.49
18	dirkaccount	425	456	↓	31	↑	6.80
19	fileappender	172	175	↓	3	↑	1.71
20	hashcode	11	32	↓	21	↑	65.63
21	lamport	372	515	↓	143	↑	27.77
22	mergesort	317	546	↓	229	↑	41.94
23	numberaxis	175	242	↓	67	↑	27.69
24	peruserpooldatasource	7	8	↓	1	↑	12.50
25	peterson	28	37	↓	9	↑	24.32
26	pingpong	28	28	–	0	–	0.00
27	pool	9	9	–	0	–	0.00
28	rax	31	31	–	0	–	0.00
29	readerswriters	1,324	1,339	↓	15	↑	1.12
30	replicatedcasestudies	104	104	–	0	–	0.00
31	rvexample	76	92	↓	16	↑	17.39
32	sharedobject	3	8	↓	5	↑	62.50
33	sharedpooldatasource	4	5	↓	1	↑	20.00
34	simple	12	18	↓	6	↑	33.33
35	store	23	28	↓	5	↑	17.86
36	transfer	2	2	–	0	–	0.00
37	waitnotify	1,159	1,456	↓	297	↑	20.40
38	Weblech	1,124	2,297	↓	1,173	↑	51.07

to the total number of times of constraint solving of J-MCR. GC-MCR constraint-solving time and J-MCR constraint-solving time represent the total time taken by the constraint solver in detecting the first concurrent bug of a test program with GC-MCR and J-MCR, respectively. The reduced solving time represents the reduced time of a constraint solver in constraint solving with GC-MCR, compared with the total time of the solver in constraint solving with J-MCR. The percentage of reduced solving time represents the percentage of reduced solving time to the total constraint-solving time of J-MCR. GC-MCR constraint-filtering time represents the total time consumed in constraint filtering and reduction required by GC-MCR to detect the first concurrent bug of a test program. The percentage of filtering-solving time represents the percentage of GC-MCR constraint-filtering time to J-MCR constraint-solving time.

According to Tables 2 and 3, firstly, compared with J-MCR, GC-MCR has a smaller number of times of constraint solving in most test programs. Using GC-MCR can reduce the number of times of constraint solving by about 20% on average, with a maximal reduction of 71.43%.

Table 3 Comparison of constraint-solving time between GC-MCR and J-MCR

ID	Test program	GC-MCR constraint-solving time (ms)	J-MCR constraint-solving time (ms)	Reduced solving time (ms)	Percentage of reduced solving time (%)	GC-MCR constraint-filtering time (ms)	Percentage of filtering-solving time (%)
1	account	794	1,029	↓ 235	↑ 22.84	15	1.46
2	airline	304,448	360,345	↓ 55,897	↑ 15.51	246	0.07
3	alarmclock	3,658,195	3,880,957	↓ 222,762	↑ 5.74	3,206	0.08
4	allocation	2,278	2,646	↓ 368	↑ 13.91	60	2.27
5	array	56,899	98,283	↓ 41,384	↑ 42.11	272	0.28
6	bakery	8,860	21,655	↓ 12,795	↑ 59.09	95	0.44
7	boundedbuffer	1,011	1,373	↓ 362	↑ 26.37	38	2.77
8	bubblesort	48,876	62,142	↓ 13,266	↑ 21.35	443	0.71
9	checkfield	289	757	↓ 468	↑ 61.82	21	2.77
10	clean	7,236	11,049	↓ 3,813	↑ 34.51	64	0.58
11	consistency	544	896	↓ 352	↑ 39.29	29	3.24
12	counter	28,457	33,786	↓ 5,329	↑ 15.77	82	0.24
13	critical	172	232	↓ 60	↑ 25.86	12	5.17
14	cyclic	23,573	27,273	↓ 3,700	↑ 13.57	70	0.26
15	datarace	353	1,158	↓ 805	↑ 69.52	31	2.68
16	deadlock	93	117	↓ 24	↑ 20.51	7	5.98
17	dekker	77,291	154,573	↓ 77,282	↑ 50.00	70	0.05
18	dirkaccount	101,272	216,110	↓ 114,838	↑ 53.14	1,098	0.51
19	fileappender	21,218	29,786	↓ 8,568	↑ 28.77	220	0.74
20	hashcode	970	2,374	↓ 1,404	↑ 59.14	36	1.52
21	lamport	22,329	38,688	↓ 16,359	↑ 42.28	86	0.22
22	mergesort	245,428	347,597	↓ 102,169	↑ 29.39	11,404	3.28
23	numberaxis	188,382	265,010	↓ 76,628	↑ 28.92	3,593	1.36
24	peruserpooldatasource	1,040	3,644	↓ 2,604	↑ 71.46	17	0.47
25	peterson	10,905	17,111	↓ 6,206	↑ 36.27	34	0.20
26	pingpong	8,918	13,409	↓ 4,491	↑ 33.49	28	0.21
27	pool	514	622	↓ 108	↑ 17.36	14	2.25
28	rax	11,302	16,625	↓ 5,323	↑ 32.02	58	0.35
29	readerswriters	220,750	322,420	↓ 101,670	↑ 31.53	2,823	0.88
30	replicatedcasestudies	16,219	29,711	↓ 13,492	↑ 45.41	133	0.45
31	rvexample	4,512	7,548	↓ 3,036	↑ 40.22	30	0.40
32	sharedobject	224	562	↓ 338	↑ 60.14	15	2.67
33	sharedpooldatasource	477	1,244	↓ 767	↑ 61.66	19	1.53
34	simple	689	1,580	↓ 891	↑ 56.39	22	1.39
35	store	2,165	3,037	↓ 872	↑ 28.71	46	1.51
36	transfer	153	194	↓ 41	↑ 21.13	16	8.25
37	waitnotify	108,400	179,769	↓ 71,369	↑ 39.70	142	0.08
38	Weblech	207,126	406,561	↓ 199,435	↑ 49.05	497	0.12

Moreover, for test programs with fewer constraint solving under J-MCR, the proportions of the reduced number of times of constraint solving after using GC-MCR are higher. Secondly, as GC-MCR reduces not only the number of times of constraint solving of most test programs but also the constraint expressions of a few test programs whose number of times of constraint solving is not reduced, the constraint-solving time of GC-MCR is less than that of J-MCR. Compared with J-MCR, GC-MCR can reduce computational resources for constraint solving by about 37% on average, with a maximum reduction of 71.46%. Thirdly, the time required by GC-MCR to filter and reduce constraints is much less than the constraint-solving time reduced by using GC-MCR, maximally accounting for only 8% of J-MCR constraint-solving time. From the above, the time overhead of constraint reduction and filtering based on directed graphs is much less than that of solving conflicting constraints with the solver. Thus, filtering and reducing constraint expressions in advance can effectively reduce the number of calls of the

constraint solver to save the constraint-solving time of programs and computational resources of the constraint solver.

Further analysis shows that test programs with a large number of times of constraint solving under J-MCR can be divided into two types according to their characteristics: test programs with many threads and those with relatively few threads but many read-write events to the same variable in each thread. The former need a lot of constraint solving due to many threads and thread interleavings. When using GC-MCR, it will judge whether read-write constraints conflict after fixing the execution order of threads. Fixing the execution order of threads can help judge conflicts more completely to filter out more constraints, thus yielding a better effect on the concurrent bug detection of GC-MCR. As the latter have few threads but many read-write events to the same variable, even if the execution order of threads is fixed, order combinations of many read-write events still hold. For this reason, using GC-MCR may not necessarily filter out many constraints, resulting in an unstable effect of constraint filtering. Further research can be carried out to solve this problem.

RQ2: Can GC-MCR effectively reduce the time of concurrent bug detection and improve the efficiency of exploring the state space of concurrent programs?

In response to RQ2, this study systematically applies GC-MCR to each test program and counts the time taken to detect the first concurrent bug, so as to check the difference between the two methods. This study compares GC-MCR with J-MCR. Test results are shown in Table 4. In the table, the average execution time of GC-MCR and that of J-MCR represent the average time required to detect the first concurrent bug in a test program by using GC-MCR and J-MCR, respectively. The difference in program execution time represents the difference between the execution time of a test program when using J-MCR and that of the test program when GC-MCR is used. The percentage of performance improvement represents the percentage of difference in program execution time to the average execution time of J-MCR.

From Table 4, firstly, compared with J-MCR, GC-MCR with constraint filtering and reduction can effectively reduce the time of concurrent bug detection. It can reduce the total calling time by 34.01% on average, indicating that the overall performance of a program is improved by 34.01% on average. Secondly, Table 2 shows that the percentage of reduced constraint solving is not proportional to that of performance improvement. Thirdly, for test programs with more average execution time under J-MCR, namely, those with a larger number of times of constraint solving when J-MCR is used, the use of GC-MCR for constraint filtering and reduction will improve their performance. Fourthly, for test programs with no constraint being filtered, using GC-MCR to reduce their constraints can also reduce program execution time to improve performance. In particular, in the real-world application Weblech, the conclusions obtained in response to RQ1 and RQ2 are still valid. This further verifies the effectiveness of GC-MCR, indicating that GC-MCR is able to detect concurrent bugs in concurrent programs with large state space and is applicable to real-world application software.

The average execution time of GC-MCR consists of three parts: constraint filtering and reducing time, constraint solving time, and program scheduling time. From further analysis, according to Table 3, the time overhead of filtering and reducing constraints is much less than that of solving conflicting constraints by the solver. Thus, once there are filterable or reducible constraints, the time overhead of constraint solving can be reduced (constraint solving includes constraint filtering and reduction). Then, based on the reduction in total time of concurrent bug detection of GC-MCR, this study finds that the reduced time overhead of constraint solving is much more than variations in the time overhead of program scheduling. This means that the reduction in bug detection time mainly comes from the difference between the reduced time overhead of constraint solving and the time overhead variation of filtering and program

Table 4 Comparison of average execution time between GC-MCR and J-MCR

ID	Test program	Average execution time of GC-MCR (ms)	Average execution time of J-MCR (ms)		Difference in program execution time (ms)		Percentage of performance improvement (%)
1	account	929	1 170	↓	241	↑	20.60
2	airline	308,183	363,946	↓	55,763	↑	15.32
3	alarmclock	3,751,189	3,976,459	↓	225,270	↑	5.67
4	allocation	2,445	2,813	↓	368	↑	13.08
5	array	57,881	98,859	↓	40,978	↑	41.45
6	bakery	9,104	21,856	↓	12,752	↑	58.35
7	boundedbuffer	1,143	1,509	↓	366	↑	24.25
8	bubblesort	49,650	62,486	↓	12,836	↑	20.54
9	checkfield	400	811	↓	411	↑	50.68
10	clean	7,380	11,248	↓	3,868	↑	34.39
11	consistency	699	982	↓	283	↑	28.82
12	counter	28,581	33,977	↓	5,396	↑	15.88
13	critical	271	320	↓	49	↑	15.31
14	cyclic	23,361	27,442	↓	4,081	↑	14.87
15	datarace	595	1,641	↓	1,046	↑	63.74
16	deadlock	172	243	↓	71	↑	29.22
17	dekker	78,075	155,317	↓	77,242	↑	49.73
18	dirkaccount	102,830	218,181	↓	115,351	↑	52.87
19	fileappender	21,954	30,468	↓	8,514	↑	27.94
20	hashcode	1,233	2,557	↓	1,324	↑	51.78
21	lamport	22,791	39,056	↓	16,265	↑	41.65
22	mergesort	284,283	380,757	↓	96,474	↑	25.34
23	numberaxis	194,446	271,151	↓	76,705	↑	28.29
24	peruserpooldatasource	1,560	3,980	↓	2,420	↑	60.80
25	peterson	11,271	17,240	↓	5,969	↑	34.62
26	pingpong	9,048	13,640	↓	4,592	↑	33.67
27	pool	720	850	↓	130	↑	15.29
28	rax	11,426	16,830	↓	5,404	↑	32.11
29	readerswriters	224,668	327,420	↓	102,752	↑	31.38
30	replicatedcasestudies	16,667	30,071	↓	13,404	↑	44.57
31	rvexample	4,840	7,724	↓	2,884	↑	37.34
32	sharedobject	330	650	↓	320	↑	49.23
33	sharedpooldatasource	920	1,590	↓	670	↑	42.14
34	simple	821	1,669	↓	848	↑	50.81
35	store	2,310	3,129	↓	819	↑	26.17
36	transfer	270	330	↓	60	↑	18.18
37	waitnotify	110,780	182,196	↓	71,416	↑	39.20
38	weblech	9,296,869	17,532,083	↓	8,235,214	↑	46.97

scheduling. The reason for this result is that the time overhead of program constraint solving is much more than that of program scheduling. GC-MCR only filters and reduces constraints, without changing program scheduling, resulting in the same quantitative relationship between variations of the two as before.

RQ3: Can GC-MCR guarantee the concurrent bug detection capacity of the existing method?

In response to RQ3, this study systematically applies GC-MCR to each test program and collects the type of the first concurrent bug detected, so as to check the difference between the two methods. Test results are shown in Table 5. In this table, the number of times of GC-MCR scheduling and the number of times of J-MCR scheduling represent the number of times of scheduling required to detect a concurrent bug in a test program by using GC-MCR and J-MCR, respectively. GC-MCR concurrent bug and J-MCR concurrent bug represent concurrent bugs in a test program detected by GC-MCR and J-MCR, respectively.

Table 5 Comparison of number of times of effective scheduling between GC-MCR and J-MCR

ID	Test program	Number of times of scheduling		Concurrent bug	
		GC-MCR	J-MCR	GC-MCR	J-MCR
1	account	3	3	Atomicity	Atomicity
2	airline	434	445	DataRace	DataRace
3	alarmclock	888	915	DataRace	DataRace
4	allocation	4	4	Atomicity	Atomicity
5	array	22	22	DataRace	DataRace
6	bakery	15	15	Atomicity	Atomicity
7	boundedbuffer	1	1	Deadlock	Deadlock
8	bubblesort	10	10	DataRace	DataRace
9	checkfield	5	5	DataRace	DataRace
10	clean	2	2	DataRace	DataRace
11	consistency	4	4	Atomicity	Atomicity
12	counter	2	2	DataRace	DataRace
13	critical	2	2	Atomicity	Atomicity
14	cyclic	8	8	DataRace	DataRace
15	datarace	3	3	DataRace	DataRace
16	deadlock	2	2	Deadlock	Deadlock
17	dekker	145	145	Atomicity	Atomicity
18	dirkaccount	2	2	Deadlock	Deadlock
19	fileappender	3	3	Atomicity	Atomicity
20	hashcode	5	5	Atomicity	Atomicity
21	lamport	75	75	Atomicity	Atomicity
22	mergesort	2	2	DataRace	DataRace
23	numberaxis	17	17	Atomicity	Atomicity
24	peruserpooldatasource	3	3	DataRace	DataRace
25	peterson	5	5	Atomicity	Atomicity
26	pingpong	4	4	DataRace	DataRace
27	pool	2	2	DataRace	DataRace
28	rax	2	2	Deadlock	Deadlock
29	readerswriters	10	10	DataRace	DataRace
30	replicatedcasestudies	1	1	Deadlock	Deadlock
31	rvexample	21	21	DataRace	DataRace
32	sharedobject	3	3	DataRace	DataRace
33	sharedpooldatasource	1	1	Atomicity	Atomicity
34	simple	6	6	Atomicity	Atomicity
35	store	2	2	Atomicity	Atomicity
36	transfer	2	2	Deadlock	Deadlock
37	waitnotify	560	560	DataRace	DataRace
38	Weblech	1,096	2,024	DataRace	DataRace

According to Table 5, firstly, GC-MCR can correctly detect concurrent bugs in these test programs, and concurrent bugs detected by GC-MCR and J-MCR are the same. Moreover, both methods trigger concurrent bugs by the same thread scheduling in most test programs, indicating that compared with J-MCR, GC-MCR has no loss of capacity to detect concurrent bugs. In the case of small program state space, randomness affects the exploration of program concurrent space slightly, showing that compared with J-MCR, GC-MCR can detect concurrent bugs with the same number of program executions. Secondly, compared with J-MCR, GC-MCR triggers concurrent bugs through less thread scheduling in the test programs airline, alarmclock, and weblech. This shows that the influence of randomness on the exploration on the program concurrent space increases with the increase in the size of program state space and that the existing method has to explore redundant thread scheduling due to redundant constraints, thus reducing detection efficiency. By contrast, GC-MCR can detect concurrent bugs through fewer program executions than J-MCR, effectively filter conflicting constraints, and reduce redundant ones. In other words, conflicting constraints filtered out by GC-MCR are invalid ones, and the

reduction of redundant constraints is equally effective and reasonable. Thus, after constraint filtering and reduction, the completeness of the test framework is not reduced, and no thread interleaving sequence is missed. This indicates that GC-MCR improves program performance without missing any test thread sequence. GC-MCR can not only improve the efficiency of concurrent bug detection but also ensure the completeness of the test framework.

Further analysis shows that GC-MCR uses directed graphs to filter and reduce constraints. For constraint solving, GC-MCR chooses to either filter or reduce current constraints to obtain equivalent constraint expressions, with no operation that modifies the results of constraint solving. Thus, despite randomness in program execution, GC-MCR can still guarantee that its scheduling schemes are at most those of J-MCR. GC-MCR ensures its completeness and can guarantee the concurrent bug detection capacity of the existing method.

4 Discussion

This study proposes a method of directed graph-based representation of concurrent constraints. On this basis, it designs and implements a concurrent bug detection method, called GC-MCR concerning constructed constraints. The existing methods build concurrent-program constraints based on maximal causal models, but with no systematic research on the representation of concurrent programs. The directed graph-based representation method proposed in this study optimizes the representation of concurrent program constraints structurally. With the advantages of directed graph-based representation, unsolvable constraints can be identified during construction, which reduces the overhead of constraint solving. Moreover, constructed concurrent-program constraints can be reduced. For the same thread interleaving, GC-MCR can build more accurate and reduced concurrent program constraints, thus improving the speed of constraint solving. Therefore, under limited testing resources, the overhead of constraint solving is reduced, and resources saved in constraint solving are used to explore the state space of concurrent programs, resulting in an improvement in the efficiency of concurrent bug detection. The test results also show that GC-MCR detects bugs in concurrent programs more quickly than the existing methods. However, from the analysis of the test results, the improvement of GC-MCR on detection efficiency is not stable, with the average execution time improving by 5% to 64%. In addition, advantages in the number of times of scheduling are not obvious. In this chapter, a discussion will be carried out from three aspects: program state space, causal models, and concurrent programs.

Performance improvement of GC-MCR is affected by the state space of concurrent programs. The state space of concurrent programs increases exponentially with the increase in the number of both threads and read-write operations in such programs. Meanwhile, with the growth of the state space of concurrent programs, the number of times of constraint solving and program scheduling before the detection of concurrent bugs also increases. According to the percentage of improvement in the test results, due to the influence of the program state space, the improvement of GC-MCR in the number of times of constraint solving and the average time is not stable. Under the existing maximal causal model, more read-write operations of a program result in more effective concurrent constraints constructed. As can be seen from Table 2, the reduced number of times of constraint solving increases with the increase in the program state space, while the increase in constraint percentage slows down due to the increase in the space of effective concurrent constraints. In addition, compared with the existing method, the constraint construction method based on directed graphs designed in this study needs to re-represent and analyze constraints. Therefore, with the growth of the program state space, the overhead of the directed graph-based representation also increases accordingly. As can be seen from Table 4, due to the excessive time consumption of constraint solving, the overhead of the

directed graph-based representation is still lower than that of constraint solving by the existing method.

The number of times of scheduling is limited by maximal causal models. In this study, the representation of concurrent programs based on directed graphs is proposed. On this basis, a construction method of concurrent program constraints is put forward. Previous studies have shown that maximal causal models can maximally reduce the exploration of redundant state space of concurrent programs. Therefore, the core model for constraint construction in this study is also based on a maximal causal model. Table 5 shows that GC-MCR has no obvious advantage over the existing method in terms of effective scheduling sequences, mainly due to the use of the same maximal causal model. The advantage of GC-MCR is that the method can generate concurrent constraints conforming to maximal causal models more effectively, reduce the overhead of constraint solving, and then detect concurrent bugs more quickly.

Relationship between evaluation data and concurrent test programs. In this study, indexes used to evaluate the concurrent bug detection capacity of GC-MCR and J-MCR include the number of times of constraint solving, average execution time, constraint-filtering time, and number of times of scheduling. This study is based on maximal causal models, and the number of times of constraint solving, constraint-filtering time, and number of times of scheduling are all limited to the maximal causal model. Therefore, the three indexes are all related to the number of threads in concurrent test programs and the number of read-write operations to variables in different threads. The average execution time is not only related to the number of times of constraint solving, constraint filtering time, and number of times of scheduling but also affected by hardware configuration and execution speed. As both average execution time and number of times of scheduling can directly and comprehensively evaluate concurrent bug detection of methods, they are taken as the evaluation indexes in this study.

4.1 Analysis of factors affecting effectiveness

Factors that may affect the effectiveness of the test results in this study are mainly analyzed as follows.

Analysis of internal effectiveness. In view of the complexity of implementing the method proposed in this study, the internal factor that may affect the effectiveness of the test results is the correctness of the code implementation of this method. In order to reduce the influence of code implementation errors on the test results of this study, the related technologies used in this study all employ mature third-party frameworks, such as ASM[24] and Z3[20]. Moreover, this study refers to code implementation of the existing research results, such as MCR[18] and CovCon[25] and adopts the mode of peer review in coding, thus ensuring the correctness of code implementation to the greatest extent.

Analysis of external effectiveness. Three main factors affect the generality of the test results in this study. The first factor is the representativeness of the datasets used in this study. In order to ensure the effectiveness of conclusions, this study employs 27 concurrent datasets commonly used in the research on concurrent bug detection. Such datasets cover common types of concurrent bugs. As the constraint solver Z3 this study relies on is a CPU-intensive tool, the second factor is the consistency of the test platform. In order to reduce the influence of the test platform on the test results, all the tests are carried out on the same platform and repeated many times to eliminate the influence of possible randomness on the test results. The third factor is the representativeness of the contrast method used in this study. This study chooses to verify the effectiveness of GC-MCR in Java programs. In comparison, other concurrent test methods, such as the active test methods based on saturated coverage, Maple[26] and IDAT[27], as well as multiple concurrent test methods based on CBMC[28–30], are all proved effective in

C programs. SeqCheck[23] detects concurrent bugs in Java programs by modeling program branch information and predicting the feasibility of event sequences. However, its source code is not disclosed. Re-implementation of the above work in Java programs will be costly, and the correctness of re-implemented programs cannot be guaranteed. Moreover, J-MCR has verified its advancement and representativeness in the latest research work. Therefore, this study selects J-MCR for comparison.

Analysis of effectiveness of conclusions. The main factor affecting the effectiveness of conclusions is the reasonableness of the evaluation indexes used in the test analysis in this study. In view of this, two commonly used indexes of evaluating the capacity of concurrent bug detection in the existing research are adopted: time taken to detect the first concurrent bug and the number of times of scheduling of a concurrent program at the time of the first concurrent bug being detected. In addition, in order to measure whether GC-MCR can effectively reduce the number of times of constraint solving, this study introduces the number of times of constraint solving as an additional evaluation index to further ensure the effectiveness of the conclusions of this study.

5 Related Work

5.1 Coverage metrics for concurrent programs

Coverage criteria of concurrent programs can be used to quantify the richness of test suites (for example, whether a program has been tested fully) or provide practical guidance on test-case generation (for example, as an objective function used in a program fuzzy engine). For example, HaPSet[31] collects sorting constraints in a program and guides program testing by analyzing constraints. CovCon[25] extracts method pairs of common executions by analyzing recorded executions to generate test cases that may cover the uncovered method pairs. TSA[33] aims to generate thread scheduling to cover uncovered coverage requirements, so as to achieve high synchronous coverage of concurrent programs. ConSuite[32] statically analyzes a thread interleaving set and checks program execution records to judge whether a specific thread interleaving is covered and then uses a genetic algorithm to generate test cases that can cover more thread interleavings. Considering calling context information, AutoConTest[33] calculates coverage requirements dynamically and iteratively, generates sequential tests according to sequential coverage, and packages sequential tests into concurrent ones.

Yang *et al*.[35] proposed definition-use coverage based on full dual-path coverage. Krena *et al*.[36] proposed a method to derive new coverage metrics for testing concurrent software based on existing dynamic or static analysis methods, such as Eraser[37] and GoldiLocks[38]. These studies have extended multiple existing coverage metrics of concurrent tests, such as ConcurPairs, definition-use coverage, and synchronization pair coverage. MAP-Coverage[22] uses memory access patterns to abstract test executions, measures which memory access patterns are covered, and then generates test cases that can cover more memory access patterns. MAP-Coverage is more abstract than thread interleaving coverage and is more relevant to bugs than method-pair coverage.

5.2 Model checking and constraint solving

In the existing research on concurrent bug detection, model checking[39–41] is often used to exhaustively explore the thread scheduling space to detect potential concurrent bugs in programs. For example, CHESS[42] dynamically explores different thread scheduling of a target program in a context-bounded manner. Shacham *et al*.[40] build test cases, based on a model checker, for races reported by the lockset algorithm[9, 43, 44]. However, in the face of exponential growth of program paths and scheduling space, model checking can hardly be extended to

large-scale multithreaded programs. For this reason, Huang *et al*. proposed MCR[18], which transformed the exploration of state space of concurrent programs into constraint solving, so as to reduce the exploration of redundant state space of such programs and improve the efficiency of concurrent bug detection. Similarly, combining constraint solving with verification-condition generation, STORM[45] detected potential bugs in Windows device drivers by constructing constraints between globally stored mappings of BoogiePL programs.

Constraint solving mainly depends on SMT or SAT solvers. At present, many efforts have been made to improve the capacity of solvers[46–48], but the difficulty in constraint solving is still an urgent problem to be solved. For this reason, Wang *et al*.[46] proposed accelerating constraint solving by using decision-making abilities of neural networks based on deep reinforced learning. Different from the above work, this study proposes a directed graph-based representation method to build a directed graph of basic constraints in concurrent-program constraints. Then, according to the directed graph, it filters and reduces concurrent program constraints to improve the speed of constraint solving and reduce relevant computational resource consumption.

5.3 Bug detection of concurrent programs

Three problems are usually studied in bug detection of concurrent programs: how to improve detection efficiency, how to improve detection effectiveness, and how to reduce false detection. The classical methods for concurrent bug detection are happens-before analysis[49, 50] and lockset algorithms[9, 43, 44], which are widely used to detect bugs in concurrent programs. For example, RaceChecker[49] is a data-race checker that uses happens-before relations to reduce infeasible races before reporting potential races to be verified. Eraser[36] proposed a lockset algorithm to detect bugs in lock-based concurrent programs by monitoring each shared-memory citation and locking behavior. Lockset's improvements[9, 43, 44, 51] can reduce overhead and false detection. Lockset and happens-before analysis can also be combined[38, 52–55]. However, due to the limitations of static analysis, compared with dynamic concurrent bug detection methods, lockset algorithms and happens-before analysis often lead to false detection.

False detection means that thread interleavings irrelevant to bugs are detected as errors. Without executing programs, static detection techniques detect bugs by analyzing source code. Therefore, checkers cannot correctly determine happens-before information, leading to false detection. MCR[18] transforms the exploration of the state space of concurrent programs into constraint solving and then controls program scheduling according to each solving result. Thus, it can explore the state space of a whole program with the least number of program executions. ConRacer[56] builds a calling graph by control-flow analysis, searches for alias objects by context-sensitive alias analysis, and finally reduces false and missed detection by happens-before analysis. Maple[26] proposed an active test method based on saturated coverage, and Yue *et al*.[27] proposed a measure of test-case diversity for multithreaded programs based on Maple. Alglave *et al*.[29] proposed a kind of integer differential logic coding, verifying deployed concurrent system code with the help of bounded model checking. He *et al*.[28] proposed an ordering consistency theory applied to multithreaded program verification in SC, realizing incremental consistency checking, minimum conflict clause generation, and specialized theory propagation. According to the Event Order Graph (EOG), Yin *et al*.[30] designed an EOG-based algorithm for counterexample verification and optimized generation, and it was able to obtain small but effective optimization constraints. On this basis, they proposed an abstraction and refinement method for multithreaded C-program verification based on scheduling constraints. SeqCheck[23] models program branches and predicts the feasibility of event sequences. It first calculates an event set to determine the feasibility of a sequence, then constructs a graph to re-sort events in the set, and finally calculates sequential closures in the graph.

This study is different from the above work. It not only transforms the exploration of the state space of concurrent programs into constraint solving but also puts forward a directed-graph-based representation method to construct complete concurrent-program constraints. Then, by constraint filtering and reduction, this study reduces the overhead of constraint solving and improves the efficiency of exploring the state space of concurrent programs.

6 Summary and Prospect

This study constructs constraints based on directed graphs to accelerate constraint solving of MCR. It processes constraints before calling the constraint solver, splitting and analyzing read-write constraints. Then, conflicts are detected between these read-write constraints and program basic constraints. Conflicting constraints are filtered, and redundant ones are reduced, so as to reduce single-solving time or even the number of calls to the constraint solver and consumption of relevant computational resources. In this way, the efficiency of exploring all possible states in concurrent programs can be improved. In this study, GC-MCR is empirically studied on 38 concurrent test programs. From the test results, under the condition of ensuring the completeness of scheduling and concurrent-bug detection capacity of MCR, GC-MCR reduces time consumption of single constraint solving and the number of calls to the constraint solver, lowers the computational resource consumption of the constraint solver, and improves the efficiency of exploring the state space of test programs. It is significantly better than the existing methods.

In future research, it is necessary to add some feasibility relaxation, so as to relax specific feasibility constraints of branch events by adding feasibility relaxation-related algorithms for such events and further improve the efficiency of concurrent bug detection without affecting accuracy.

References

[1] Jiang YY, Xu C, Ma XX, *et al*. Approaches to obtaining shared memory dependences for dynamic analysis of concurrent programs: A survey. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 747–763 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5193.htm [doi: 10.13328/j.cnki.jos.005193]

[2] Bo LL, Jiang SJ, Zhang UM, Wang XY, Yu Q. Research progress on techniques for concurrency bug detection. Computer Science, 2019, 46(5): 13–20 (in Chinese with English abstract).

[3] Su XH, Yu Z, Wang TT, Ma PJ. A survey on exposing, detecting and avoiding concurrency bugs. Chinese Journal of Computers, 2015, 38(11): 2215–2233 (in Chinese with English abstract).

[4] Putchala MK, Bryant AR. Synchron-ITS: An interactive tutoring system to teach process synchronization and shared memory concepts in an operating systems course. Proc. of the 2016 Int'l Conf. on Collaboration Technologies and Systems (CTS 2016). 2016. 180–187.

[5] Huang J, Meredith PO, Rosu G. Maximal sound predictive race detection with control flow abstraction. Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2014). 2014. 337–348.

[6] Park S, Vuduc RW, Harrold MJ. Falcon: Fault localization in concurrent programs. Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering (ICSE 2010). 2010. 245–254.

[7] Lu GZ, Xu L, Yang YB, Xu BW. Predictive analysis for race detection in software-defined networks. Science China Information Sciences, 2019, 62(6): 062101:1–062101:20.

[8] Cai Y, Lu Q. Dynamic testing for deadlocks via constraints. IEEE Trans. on Software Engineering, 2016, 42(9): 825–842.

[9] Engler DR, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP 2003). 2003. 237–252.

[10] Letko Z, Vojnar T, Krena B. AtomRace: Data race and atomicity violation detector and healer. Proc. of

the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2008), Held in Conjunction with the ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2008). 2008.

[11] Huang J, Zhang C, Dolby J. CLAP: Recording local executions to reproduce concurrency failures. Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2013). 2013. 141–152.

[12] Lu S, Tucek J, Qin F, Zhou YY. AVIO: Detecting atomicity violations via access-interleaving invariants. IEEE Micro, 2007, 27(1): 26–35.

[13] Clarke EM, Grumberg O, Minea M, Peled D. State space reduction using partial order techniques. Int'l Journal on Software Tools for Technology Transfer, 1999, 2(3): 279–287.

[14] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2005). 2005. 110–121.

[15] Godefroid P. Partial-order Methods for the Verification of Concurrent Systems—An Approach to the State-explosion Problem. Springer, 1996.

[16] Mazurkiewicz A. Trace Theory. Petri Nets: Applications and Relationships to Other Models of Concurrency. Berlin, Heidelberg, 1987. 278–324.

[17] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558–565.

[18] Huang J. Stateless model checking concurrent programs with maximal causality reduction. Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2015). 2015. 165–174.

[19] Serbanuta TF, Chen F, Rosu G. Maximal causal models for sequentially consistent systems. Proc. of the 3rd Int'l Conf. on Runtime Verification (RV 2012). Revised Selected Papers, 2012. 136–150.

[20] de Moura LM, Bjørner N. Z3: An efficient SMT solver. Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), Held as Part of the Joint European Conf. on Theory and Practice of Software (ETAPS 2008). 2008. 337–340.

[21] Zhao YQ, Wang Z, Liu S, Sun J, Chen JJ, Chen X. Achieving high MAP-coverage through pattern constraint reduction. IEEE Trans. on Software Engineering, 2022. [doi: 10.1109/TSE.2022.3144480]

[22] Wang Z, Zhao YQ, Liu S, Sun J, Chen X, Lin HR. MAP-coverage: A novel coverage criterion for testing thread-safe classes. Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2019). 2019. 722–734.

[23] Cai Y, Yun H, Wang JQ, Qiao L, Palsberg J. Sound and efficient concurrency bug prediction. Proc. of the 29th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE 2021). 2021. 255–267.

[24] ASM bytecode analysis framework. http://asm.ow2.org/.

[25] Choudhary A, Lu S, Pradel M. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. Proc. of the 39th Int'l Conf. on Software Engineering (ICSE 2017). 2017. 266–277.

[26] Yu J, Narayanasamy S, Pereira C, Pokam G. Maple: A coverage-driven testing tool for multithreaded programs. Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA 2012). 2012. 485–502.

[27] Yue H, Wu P, Chen T-Y, Lv Y. Input-driven active testing of multi-threaded programs. Proc. of the 2015 Asia-Pacific Software Engineering Conf. (APSEC). 2015. 246–253.

[28] He F, Sun ZH, Fan HY. Satisfiability modulo ordering consistency theory for multi-threaded program verification. Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation (PLDI 2021). 2021. 1264–1279. [doi: 10.1145/3453483.3454108]

[29] Alglave J, Kroening D, Tautschnig M. Partial orders for efficient bounded model checking of concurrent software. Proc. of the 25th Int'l Conf. on Computer Aided Verification (CAV 2013). 2013. 141–157.

[30] Yin L, Dong W, Liu WW, Wang J. On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans. on Software Engineering, 2020, 46(5): 549–565.

[31] Wang C, Said M, Gupta A. Coverage guided systematic concurrency testing. Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE 2011). 2011. 221–230.

[32] Hong S, Ahn J, Park S, Kim M, Harrold MJ. Testing concurrent programs to achieve high synchronization coverage. Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA 2012). 2012. 210–220.

[33] Steenbuck S, Fraser G. Generating unit tests for concurrent classes. Proc. of the 6th IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST 2013). 2013. 144–153. [doi: 10.1109/ICST.2013.33]

[34] Terragni V, Cheung S-C. Coverage-driven test code generation for concurrent classes. Proc. of the 38th Int'l Conf. on Software Engineering (ICSE 2016). 2016. 1121–1132. [doi: 10.1145/2884781.2884876]

[35] Yang CD, Souter AL, Pollock LL. All-du-path coverage for parallel programs. Proc. of the ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA'98). 1998. 153–162.

[36] Krena B, Letko Z, Vojnar T. Coverage metrics for saturation-based and search-based testing of concurrent software. Proc. of the 2nd Int'l Conf. on Runtime Verification (RV 2011). LNCS 7186, 2012. 177–192.

[37] Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems, 1997, 15(4): 391–411.

[38] Choi JD, Lee K, Loginov A, O'Callahan R, Sarkar V, Sridharan M. Efficient and precise datarace detection for multithreaded object-oriented programs. Proc. of the 2002 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2002). 2002. 258–269. [doi: 10.1145/512529.512560]

[39] Musuvathi M, Qadeer S, Ball T, Basler G. Finding and reproducing heisenbugs in concurrent programs. Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2008). 2008. 267–280.

[40] Shacham O, Sagiv M, Schuster A. Scaling model checking of dataraces using dynamic information. Journal of Parallel and Distributed Computing, 2007, 67(5): 536–550.

[41] Khurshid S, Pasareanu CS, Visser W. Test input generation with Java PathFinder: Then and now (invited talk abstract). Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2018). 2018. 1–2.

[42] Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI 2007). 2007. 446-455.

[43] Nishiyama H. Detecting data races using dynamic escape analysis based on read barrier. Proc. of the 3rd Virtual Machine Research and Technology Symp. 2004. 127–138.

[44] Elmas T, Qadeer S, Tasiran S. Precise race detection and efficient model checking using locksets. Technical Report, MSR-TR-2005-118, Microsoft Research Microsoft Corporation, 2005.

[45] Lahiri SK, Qadeer S, Rakamaric Z. Static and precise detection of concurrency errors in systems code using SMT solvers. Proc. of the 21st Int'l Conf. on Computer Aided Verification (CAV 2009). 2009. 509–524.

[46] Wang F, Rompf T. From gameplay to symbolic reasoning: Learning SAT solver heuristics in the style of Alpha (Go) zero. arXiv:1802.05340v1, 2018.

[47] Bello I, Pham H, Le QV, Norouzi M, Bengio S. Neural combinatorial optimization with reinforcement learning. Proc. of the 5th Int'l Conf. on Learning Representations (ICLR 2017). 2017.

[48] Xu L, Hoos H, Leyton-Brown K. Predicting satisfiability at the phase transition. Proc. of the 26th AAAI Conf. on Artificial Intelligence. 2012.

[49] Lu K, Wu Z, Wang XP, Chen C, Zhou X. RaceChecker: Efficient identification of harmful data races. Proc. of the 23rd Euromicro Int'l Conf. on Parallel, Distributed, and Network-based Processing (PDP 2015). 2015. 78–85.

[50] Perkovic D, Keleher PJ. Online data-race detection via coherency guarantees. Proc. of the 2nd USENIX Symp. on Operating Systems Design and Implementation (OSDI 1996). 1996. 47–57.

[51] von Praun C, Gross TR. Object race detection. Proc. of the 2001 ACM SIGPLAN Conf. on Object-

oriented Programming Systems, Languages and Applications (OOPSLA 2001). 2001. 70–82.

[52] Yu Y, Rodeheffer T, Chen W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. Proc. of the 20th ACM Symp. on Operating Systems Principles (SOSP 2005). 2005. 221–234.

[53] von Praun C, Gross TR. Static conflict analysis for multi-threaded object-oriented programs. Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI 2003). 2003. 115–128.

[54] Elmas T, Qadeer S, Tasiran S. Goldilocks: A race and transaction-aware Java runtime. Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI 2007). 2007. 245–255.

[55] Pozniansky E, Schuster A. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. Concurrency and Computation Practice and Experience, 2007, 19(3): 327–340.

[56] Zhang Y, Liu H, Qiao L. Context-sensitive data race detection for concurrent programs. IEEE Access, 2021, 9: 20861–20867.

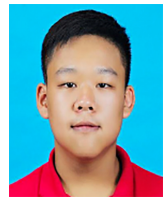
Shuochuan Li, master's degree candidate. His research interests include concurrent bug detection and system software analysis.

Yingquan Zhao, Ph.D. candidate. His research interests include compiler testing, JVM testing, and concurrency testing.

Zan Wang, Ph.D., associate professor. His research interests inlude software testing and machine learning.

Haichi Wang, Ph.D. candidate. His research interest is system software analysis.

Mingxu Ma, bachelor. His research interest is concurrency testing.



Haoyu Wang, master's degree candidate. His research interests include concurrent bug detection and compiler testing.

Xiang Chen, Ph.D., associate professor, master's supervisor. His research interests include intelligent software engineering, software warehouse mining, and software testing and maintenance.