Efficient Execution Path Exploration for Detecting Races in Concurrent Programs

Theodorus E. Setiadi, Akihiko Ohsuga, and Mamoru Maekawa

***Abstract*—Concurrent programs are more difficult to test or debug than sequential programs because their non-deterministic behaviors can produce errors that depend on timing and interleaving of threads. A different interleaving might affect branch outcomes that can lead the execution path into one different from that in which the error was detected. In order to detect concurrent errors, a programmer needs to re-execute the concurrent program many times by changing the interleaving, but it is not always feasible to conduct all the tests due to a large number of possible different interleavings. This paper proposes an efficient method to minimize the number of test cases for detecting errors in a concurrent program. This method generates test cases with different interleavings based on the execution trace. The method reduces redundant test cases without sacrificing the precision of error detection. The method is novel because it exploits the branch structure and utilizes data flows from trace information to identify only those interleavings that affect branch outcomes, whereas existing methods try to identify all interleavings that seem to affect shared variables. In order to reduce the number of test cases, those execution paths with equivalent lock sequences and accesses to shared variables are grouped together into the same "race-equivalent" group and only one member of the group is tested. We evaluated the proposed method against several concurrent Java programs. The experimental results for a Java program for telnet show the number of test cases is reduced from 147, which is based on the existing TPAIR method, to only 2 by the proposed method. Moreover, for concurrent programs that contain infinite loops, the proposed method generates only a finite and very few number of test cases, while many existing methods generate an infinite number of test cases.**

***Index Terms*— race detection, testing, concurrent program**

I. Introduction

A. Background

CONCURRENT programs are difficult to test or debug because their non-deterministic behaviors can produce errors that depend on timing, such as race conditions. It is suggested that race conditions occur mostly because shared variables are accessed by threads using inconsistent locking or even no locks[1]–[5]. Programmers often fail to apply appropriate locks due to difficulties in predicting the execution path or interrupt timing because of the complexity of concurrent programs, especially when branches are affected by access to shared variables and interleavings. To detect race conditions, a programmer can execute the concurrent program and check the execution trace using a dynamic race detector. Unfortunately, concurrent errors might not be easy to detect because a re-executed concurrent program might execute with a different interleaving. Adding additional commands or instrumentation of the source code to record intermediate results for testing concurrent programs might change the interleaving, so that errors may not show up. Unfortunately again, dynamic race detectors can detect potential errors only if they show up in a re-execution.

In this paper, we propose a new, efficient dynamic method to minimize the number of test cases for detecting concurrent errors. This is an improvement over the existing method [11]. Our proposed method iteratively uses previous execution traces as guidance for generating new test cases. The method is particularly intended for situations in which concurrent errors are difficult to detect. The number of executions needed for testing is the number of possible interleavings of the concurrent program. Even when the input values are fixed, the number of executions is still very large. The main problem is how to reduce this number of re-executions.

The contributions of this paper are as follows:

- Eliminating redundant test cases: The proposed method reduces the number of interleavings to be tested by exploiting the branch coverage information from the execution trace. This method is different from previous methods because it can distinguish those interleavings that can affect branch outcomes from those that cannot. The existing reachability testing algorithms try to identify all interleavings which may affect shared variables, although they may not necessarily affect branch outcomes; thus redundant interleavings are included. These redundant interleavings are, however, reduced in our method, resulting in a significant reduction in the number of interleavings for testing.
- Eliminating infeasible test cases: The existing reachability testing algorithms do not consider the synchronization event dependency of the execution path, e.x. lock-unlock and wait-notify mechanisms. There exist infeasible interleavings due to this dependency. The proposed method extends the existing model of variant graphs to identify infeasible interleavings due to this dependency, thereby further contributing to reducing the number of test cases.

Manuscript received August 25, 2012; revised June 22, 2013.

T. E. Setiadi is with the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. The author was supported by the JINNAI international student scholarship. (phone: +81-90-4171-9071; e-mail: eric@maekawa.is.uec.ac.jp).

A. Ohsuga is with the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. (e-mail: akihiko@ohsuga.is.uec.ac.jp).

M. Maekawa is with the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. (e-mail: maekawa@maekawa.is.uec.ac.jp).

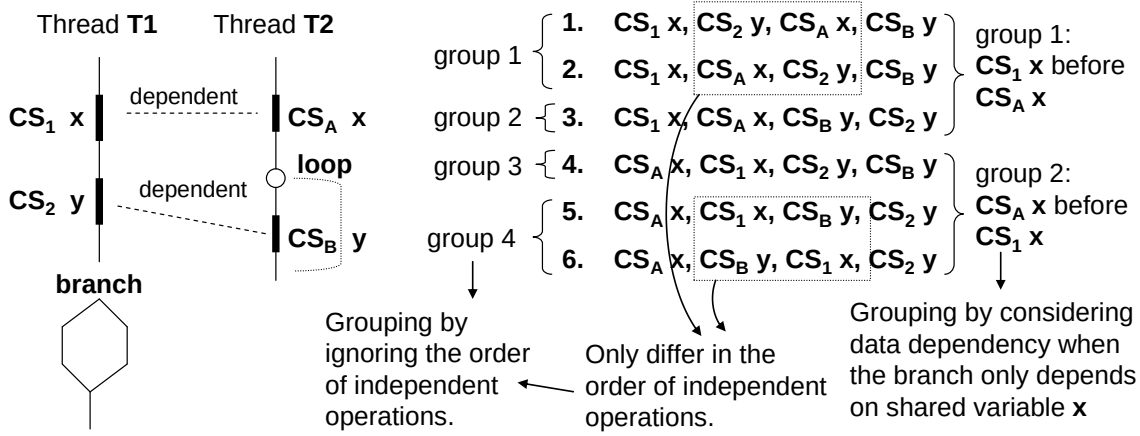


Fig. 1. Example of grouping for interleaving.

B. Motivation

Several methods have been proposed to reduce the number of interleavings that need to be tested. Partial order reduction is a general method which considers only those interleavings that may affect an execution of a program based on certain criteria. One example of the partial order reduction method reduces the number of interleavings by considering only those that may affect the values of shared variables [12] [13] [14] and by ignoring the order of independent operations, where two operations are said to be independent if any different order of the operations does not affect the values of shared variables. An example of independent operations is two read operations from different threads accessing the same shared variable. Such interleaving is left unordered because its order is irrelevant to the resulting values of any shared variables. Unfortunately, such partial order reduction still leaves some redundancy when exploring different execution paths in a thread for detecting potential race conditions. Consider the example in Fig. 1. In the case that the loop in the thread *T2* is executed only once, there are six possible different interleavings. The first and the second interleavings are different only in the order of independent operations, so they will have the same values for shared variables. A similar situation happens for the fifth and sixth interleavings. By ignoring the order of independent operations, there will be only four groups of interleavings with different combinations of values for the shared variables *x* and *y*. For members of the same group, the same read or write operation is guaranteed to use the same value of shared variable. If the branch only depends on the shared variable *x*, there are actually only two groups that matter for changing the execution path of thread *T1*. These groups are determined by whether $CS_1\ x$ is executed before $CS_A\ x$ (group 1) or vice versa (group 2). When the loop in the thread *T2* is executed several times or possibly becomes an infinite loop, there are more possible interleavings that affect the value of the shared variable *y*, but still there are only two groups of interleavings with respect to different values of the shared variable *x*. We will use this idea for exploring different execution paths efficiently.

Fig 2 shows some possible execution paths for an execution of a concurrent program. A thread can take a different execution path with a different lock sequence or different accesses to shared variables. To detect the concurrent errors, we need to find all different interleavings that can change the execution path.

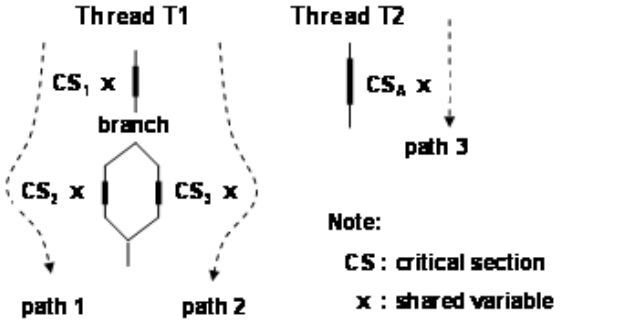


Fig. 2. Example of some possible execution paths for a concurrent program.

Suppose that path 1 is executed concurrently with path 3 (path 1 ‖ path 3) when the program is first tested. In this case, there are three possible different interleavings:

1. $CS_A\ x$, $CS_1\ x$, $CS_2\ x$
2. $CS_1\ x$, $CS_A\ x$, $CS_2\ x$
3. $CS_1\ x$, $CS_2\ x$, $CS_A\ x$

Referring to Fig 2, let us assume that the first interleaving is taken when the program is first tested. The other two interleavings are other possible test cases. Assuming that the branch is conditioned by the shared variable *x*, the conditional statement of the branch is only affected by the order of $CS_A\ x$ and $CS_1\ x$. In this example, $CS_1\ x$ is executed before $CS_A\ x$ in the second and the third interleavings, so they will result in the same execution path for thread *T1*, that is either path 1 or path 2. Since thread *T1* follows the same execution path in the second and third interleavings, this will not change the consistent locking among threads. In other words, the same lock sequences and accesses to shared variables are held. If the branch condition is *true*, then both the interleavings will lead to the execution of path 1 concurrent with path 3 (path 1 ‖ path 3). On the other hand, if the branch condition is *false*, then we will have the combination of the execution of path 2 concurrent with path 3 (path 2 ‖ path 3). For exploring different execution paths in thread *T1* caused by the branch, we need only test one of the interleavings, that is either the second or third interleaving. By considering the dependency between the conditional statement in the branch and the shared variables, we can avoid testing interleavings that do not change the execution path of a thread. For the example in Fig 2, if we know from the previous executions that the branch is not affected by the

shared variable x, then there is no need to test the second or the third interleaving. Of course, the final result for the value of the shared variable x can be different in the second and third interleavings because it might also depend on the order of CS_A x and CS_2 x.

If the execution path in thread *T1* changes to path 2, we compare the locking for accesses to shared variables between CS_2 and CS_3 before checking the race conditions for the concurrent execution of path 2 and path 3 (path 2 ‖ path 3). If the locking for accesses to shared variables in CS_2 and CS_3 is the same or "equivalent", then the race conditions are the same as in the first test case (path 1 ‖ path 3) in the previous execution, thus reducing the effort for checking race conditions.

II. RELATED WORK

Error detection can be classified into static and dynamic methods. Static methods only employ source code analysis at compile time without executing the program. Since static methods do not know the precise execution of a program that causes the error, they use a conservative approach by considering all possible executions in order not to overlook potential errors [39]. Static methods are more suitable for testing because they check all possible program behaviors. For debugging, static methods often suffer from the detection of false positives; that is, potential race conditions that do not actually exist in the execution.

Dynamic methods [15]–[17] execute the programs and detect errors using information from the execution of the program, including the execution trace and source code information. For the purposes of debugging, dynamic methods are more suitable than static methods because they can perform more precise error detection by employing the execution trace. There are some execution trace analysis techniques that use lockset analysis [1] [16] [37] for dynamically detecting race conditions. They verify whether an execution of a program satisfies a locking discipline. For example, Eraser [1] is a lockset analysis that identifies a race condition from a particular execution by checking the consistency of locking for accesses to shared variables. Most research in this field is focused on reducing false positives [15] [35] [39] [43] and reducing the overheads caused by tracing [30] [38]. J. Huang, J. Zhou, and C. Zhang [38] identified one of the causes of redundancy to be that an execution trace often contains a large number of events that are mapped to the same lexical statements in the source code. However, removing them without careful analysis might cause false negatives because they might affect the reproduction of race conditions. This situation happens when a number of events from the same lexical statement in the source code affect a conditional statement in a branch whose"then" and "else" statements have a different lock sequence and accesses to shared variables.

In other work by C. Park, K. Sen, P. Hargrove, and C. Iancu [40], known as active testing, imprecise dynamic analysis of an execution trace is performed to generate a set of tuples that represents potential concurrent errors. In the later phase, it re-executes the program by actively controlling the thread schedule to confirm the concurrent errors. However, there might be some false negatives for detecting race conditions because the set of tuples might be incomplete if some tuples were not executed in the execution. This situation happens when a race condition is caused by the "then" or "else" statements of a branch whose conditional statement is affected by interleaving. Race conditions can only be detected using dynamic methods if the execution trace contains the potential concurrent errors. Unfortunately in a concurrent program, a branch can take a different execution path not only due to different input values, but also due to different interleavings. Hence, depending on the branches and interleavings, an execution trace might or might not contain potential race conditions.

Deterministic replay techniques are available for replaying an execution of a concurrent program with the same interleaving. Such techniques record the concurrent execution trace in a recording mode. The recorded execution can be replayed later in a replaying mode for dynamic analysis. A commercial tool for deterministic replay [27] is capable of reproducing the original execution order of threads, thus the same interleaving can be replayed. When a concurrent error is detected during a recording mode, a deterministic replay requires only one execution to replay the error and obtain the execution trace containing the error. This is useful for debugging concurrent programs. However, this is only effective if programmers can identify the errors when a concurrent program is running in recording mode during software development or a testing cycle. Unfortunately, due to the huge number of all possible interleavings, not all of them can be tested during software development or the testing cycle because of time and cost restrictions. Sometimes only regression tests are performed after fixing bugs and the software is quickly deployed in real situations, leaving the possibility that other errors remain. In recording mode, all the information necessary for replaying can be traced using instrumentation [7] or a specialized virtual machine [6]. Hence, programs run more slowly during recording mode and require more memory to store information about interleaving and program states. This is known as the probe effect. Therefore, executions cannot always be traced during the deployment of systems that require high performance or where resources are limited, such as in embedded systems. To reduce the probe effect, a special hardware device can be used to communicate with the performance monitor through JTAG (refer to IEEE 1149) for tracing, but many hardware constructions cannot run at full speed when JTAG is used [33]. The advantage of this approach is that an execution can be traced with minimum interference, but the drawback is expensive hardware costs.

In cases when an error has happened in the absence of a complete execution trace for replaying, programmers need to test the concurrent program with tracing to see if the same error can be detected. Unfortunately, the error might not be easy to detect because a concurrent program can have a different interleaving during re-execution. In this situation, programmers need to control the interleaving and use deterministic testing. Deterministic testing can enforce a particular interleaving specified in test cases. Since the number of possible different interleavings can be huge, the method proposed in this paper helps in the efficient

generation of test cases to reproduce the same or equivalent execution conditions.

Some tools for deterministic replay can also be used for deterministic testing. For example, in Jreplay [7] programmers can control the interleaving by enforcing thread switching using some additional locks, and can write them in the locations where a thread switch should occur. Enforcing a thread switch is realized by unblocking the next thread in the schedule followed by blocking all other threads, including the current thread. An additional lock object is assigned to each thread. The wait and notifyAll methods are used to implement the block and unblock operations that suspend and resume an execution of a thread. A binary semaphore is used to prevent deadlocks in the control transfer method due to interceptions by the JVM scheduler. Another method devised by Pugh and Ayewah [36] uses a clock to synchronize the order of executions in multiple threads. Programmers can delay operations within a thread until the clock has reached a desired tick.

Determining which of all the possible interleavings are necessary is important because it directly affects the efficiency of test case generations. Basically, there are three approaches:

1. Random: inspect only some of all the possible interleavings using randomizing or noise injection.
2. Partial: inspect only some of all possible interleavings based on certain criteria.
3. Exhaustive: inspect all possible interleavings.

Random approaches might not discover errors because only some of the possible execution paths are inspected. An improved random approach uses a heuristic approach [18] to reduce the search space. Another improvement - carried out in ConTest - uses coverage to guide the heuristic test generation [2], but still does not ensure that errors will be found because not all possible execution paths are tested. Basically, finding errors requires an exhaustive approach. Unfortunately, exhaustive approaches often suffer from an explosion of the number of possible execution paths to be inspected. The idea behind a partial approach is to identify a group of execution paths with the same coverage. For each particular group, it is sufficient to test only one group. In program testing, the "coverage" criterion states how much of the program execution space is to be covered during testing. We can identify five levels of criteria based on program structure [8]. These are statement coverage, node coverage, branch coverage, multiple condition coverage, and path coverage. Statement coverage and node coverage are rather weak criteria, representing necessary but by no means sufficient conditions for conducting a reasonable test. Branch coverage and multiple condition coverage are stronger criteria. Path coverage is the most thorough of all, and it is necessary to ensure the correctness of a program by testing or to find errors in debugging. However, it is normally difficult to achieve, particularly in a concurrent program, because the number of possible execution paths might be huge. Nevertheless, since different execution paths might exercise different lock sequences and accesses to shared variables that can affect consistent locking, it is necessary to adopt path coverage to ensure that all concurrent accesses to shared variables are consistent.

In the field of concurrent programs, there exist some other criteria besides structural coverage that can help to determine which interleavings should be tested. For example, CHESS [19] generates all interleavings of a given scenario written by a tester based on fair scheduling. Another approach [20] exhaustively generates all possible execution orders for test cases for the purpose of mitigating memory consumption problems by dynamically building partitions along the traces. There is also a coverage model for evaluating concurrent completeness. Synchronization coverage [9] covers different orders of synchronization events from different threads. Its goal is to check whether the synchronization statements have been properly tested. For example, the *tryLock* method of the Lock interface in Java 1.5 is used to check whether a lock is available. It does not block, but may succeed or fail depending on whether another thread is holding the lock. If it always succeeds or always fails, then either the tests are sufficient or the operations are redundant.

Another approach uses program flow as a coverage criterion for examining an execution of a program. All-du-path coverage [10] uses define-use associations and is applicable for parallel programs. Synchronization coverage and All-du-path coverage are not suitable criteria for checking consistent locking among threads: consider cases where only the order of the try-locks is different for synchronization coverage or there are different define-use associations for All-du-path coverage, but lock consistency is not changed. In these cases, all possible concurrent lock-unlock sequences and accesses to shared variables may not be covered. The work done by Koushik Sen and Gul Agha [44] [45] is intended to facilitate the exploration of different execution paths. Their tool, called "JCute", explores execution paths by generating new interleavings as well as new input. It generates all possible interleavings based on previous executions by changing the order of thread executions, starting from the smallest indexed thread. Redundancy is still present here, because not all interleavings would change branch execution and locking, as previously shown by the example in the *Motivation* subsection

III. OVERVIEW OF THE EXISTING REACHABILITY TESTING METHOD

This section explains an existing method for generating test cases for concurrent programs using the reachability testing method [11] [21] [22]. This is a dynamic method that uses partial order reduction for reducing test cases. The reachability testing method in [11] covers all different interleavings that affect the values of shared variables as test cases. This reachability testing uses the previous execution trace to derive different read-write sequences that affect values of shared variables. Assume that *S* is a read-write sequence from an execution of a concurrent program. The concept of reachability testing is defined as follows:

1. Use *S* to derive other read-write sequences, called "execution-variants", that produce different values of shared variables.
2. Perform deterministic testing based on the result from step 1 using tracing.

3. For each new "execution-variant" from step 2, repeat step 1 and 2 until no more "execution-variants" are found.

Variant Graph

The reachability testing method performs an efficient exploration of "execution-variants" by grouping and ignoring different interleavings that do not affect values of shared variables, using the idea of partial order reduction. Test cases are generated systematically using a variant graph. A variant graph derives different read-write sequences from the previous execution trace. A different read-write sequence that affects the values of shared variables is called an "execution-variant". "Execution-variants" are used as test cases in reachability testing. Algorithm 1 shows how to create a variant graph from an execution trace of a concurrent program.

Algorithm 1. Creating a variant graph.

Definitions:

- *S(j)* is a read-write sequence for thread *Tj*.
- *S(j, i)* is the *i*-th operation in the sequence of thread *Tj*.

Each node *N* in the "execution-variant" graph contains the following two vectors:

- index vector: (id_1, id_2, … , id_p), where *p* is the number of threads and id_j indicates the *i*-th operations in a thread *Tj* when node *N* is generated. The index vector is initialized to zero and increased by one after each read or write operation in the thread *Tj*.
- version vector: (ver_1, ver_2, …, ver_q), where *q* is the number of shared variables and ver_k is the version number of variable *Vk* when node *N* is generated. The version for variable *Vk* is initialized to zero and increased by one after each write operation to the variable *Vk*.

Input: read-write sequence.

Output: variant graph.

Step 1. Initialize the variant graph.

Create an initial node and label it as "unmarked". Set its index vector to (0,0, … , 0) and version vector to (0,0, … , 0).

Step 2. Derive different read-write sequences.

2.1 Select an "unmarked" node, say *N*.

For each *j*, $1 \le j \le p$, where *p* is the number of threads

If id_j < the length of *S(j)*,

Then construct a child node *N'* of *N* according to steps 2.2 – 2.5.

2.2 Set the index vector of *N'* to that of *N* except that the *j*-th element is $id_j + 1$.

2.3 Set the version vector of *N'* to that of *N*.

2.4 Let var_k be a shared variable in the operation *S(j, id_j +1)* and ver_k is the version number of variable var_k in *S(j, idj +1)*.

2.5 **If** *S(j, idj +1)* is a write operation to shared variable var_k,

Then increase the ver_k' of *N'* by 1.

Step 3. Identify an "execution-variant".

3.1 Let ver_k' be the *k*-th element of the version vector of *N'*.

3.2 **If** ver_k != ver_k'

Then label *N'* as "marked" and "execution-variant" (*V*).

Else If the variant graph already contains a node with the same index and version vector as *N'*.

Then label *N'* as "marked"

Else label *N'* as "unmarked"

Step 4. Repeat **step 2** until all nodes in it are labeled "marked". Do not create child nodes for the nodes which are labeled as "execution-variant" (*V*), as this will be done later by executing them as test cases.

Note that we first need to identify all shared variables from source code before creating a variant graph. If we do not consider all shared variables, then later we might need to reconstruct the variant graph when other variables are found to be shared. It is not enough just to identify shared variables from the execution trace because maybe not all shared variables can be detected from a particular execution trace. Unfortunately, it is not always possible to identify precisely all shared variables from source code: in the case that threads are dynamically created according to input data, for example, it is necessary to consider all potential shared variables. If some variables are not actually shared, they will lead to redundant nodes in a variant graph, but they will not produce redundancy in test cases because they will not lead to any new "execution-variants".

Model for Concurrent Program Execution Traces

A concurrent program execution trace contains a sequence of operations from all the threads. An operation in a thread is modeled as a triplet of:

location : *operation* : *operand*, where

- *location* is *thread_name*:*file_name*:*line_of_code*. The thread name or the file name is omitted in some cases for simplicity when there is no ambiguity.
- *operation* is the **read** or **write** operation on a shared variable.
- *operand* is the name of the shared variable.

Fig 3 shows an example of a concurrent program and its flow graph. Let us assume that the following read and write sequence *S* is obtained from an execution trace of the first test:

T1:1 read *x*, *T2*:10 write *x*, *T1*:1 read *y*, *T1*:1: write *n*, *T1*:2 read *n*, *T2*:11 write *y*, *T1*:3 …, *T1*:7 read *y*, *T2*:12 read *x*.

Fig.4 is an example of a variant graph constructed using Algorithm 1 for the execution trace above. Lined boxes in a variant graph represent possible read-write sequences where they access the same values of the shared variables as in the previous execution. A dotted box in a variant graph represents an "execution-variant" (*V*) in which some read or write operations access values of shared variables different from the previous execution as a result of a different interleaving. There are seven "execution-variants" *V1*, *V2*, *V3*, *V4*, *V5*, *V6*, and *V7* in Fig. 4.

Fig 4 shows two equivalent read-write sequences surrounded by dotted lines. They are equivalent in terms of the read-write sequence, in the sense that every operation will read or write the same versions of shared variables. The reachability testing algorithm [11] performs reduction by only considering one of them as an "execution-variant".

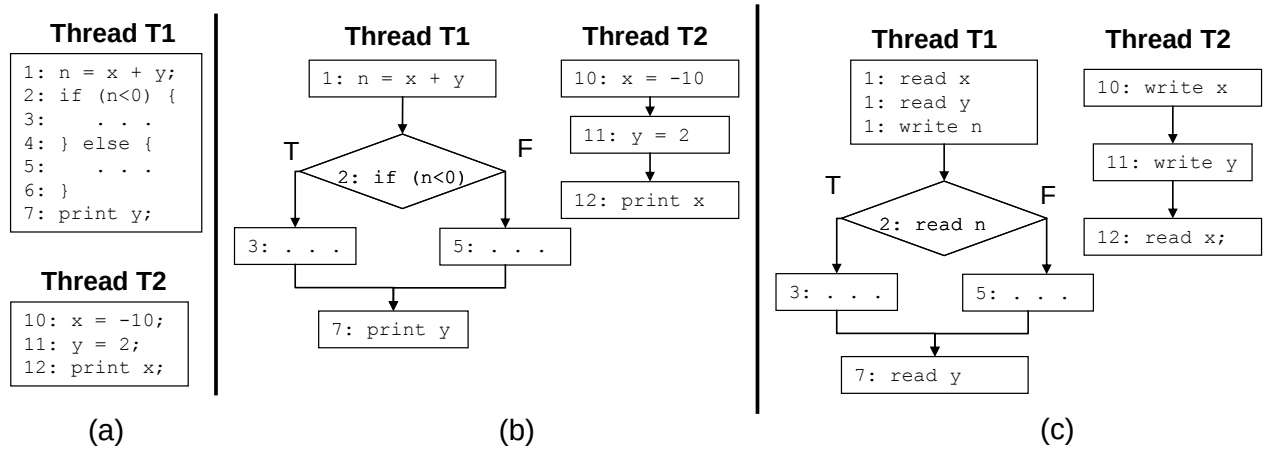


Fig. 3. (a) Example of a concurrent program (b) Flow graph. (c) Flow graph for read and write operations.

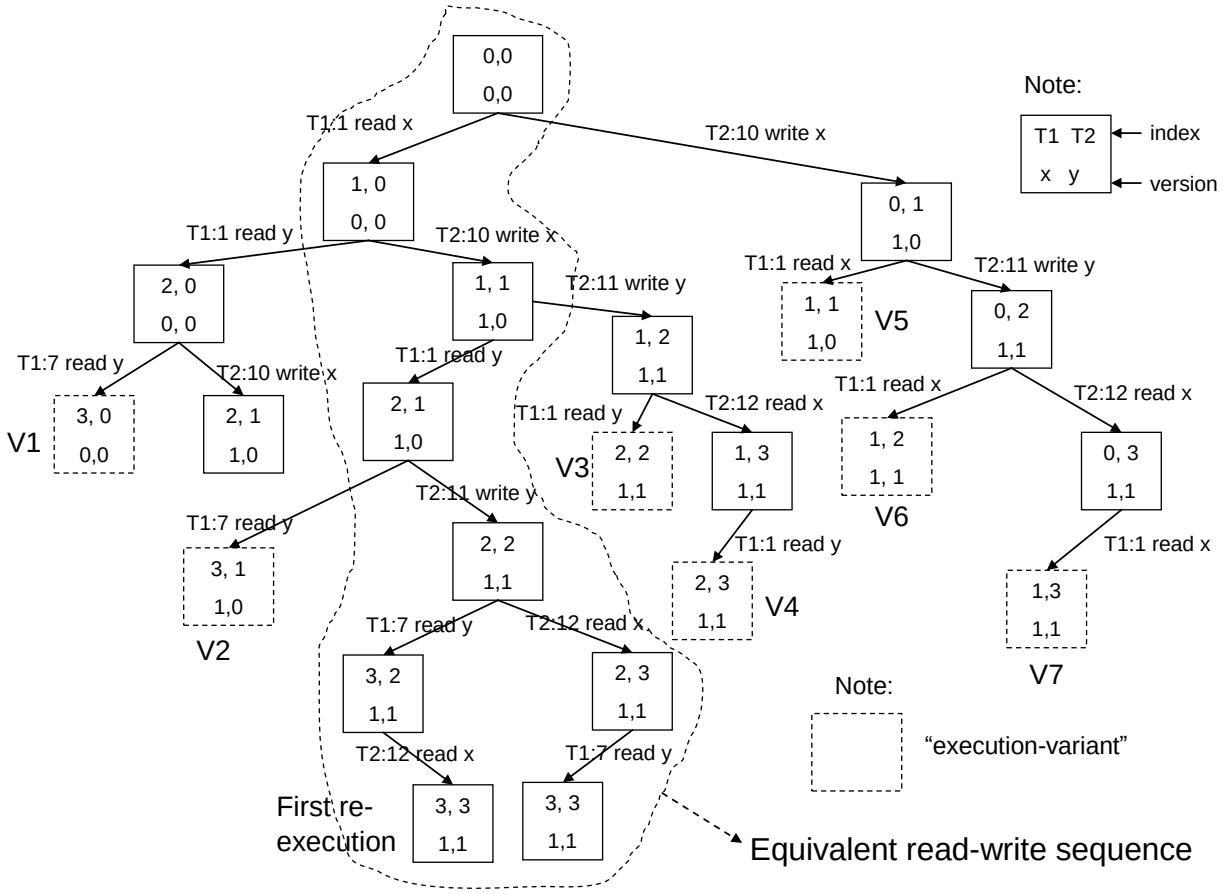


Fig. 4. Example of a variant graph from an execution trace.

IV. PROPOSED METHOD

First, the following essential terms are defined.

A. Terms

Execution Path

A concurrent program consisting of threads *T1*, *T2*, *T3*, …, *Tp*, where *p* is the number of threads. An execution path is defined in the scope of a thread and a concurrent program:

- An execution path *Pi* of a thread *Ti* is a sequence of operations executed by the thread *Ti*. For the execution of the program shown in Fig. 6(a) and Fig. 6(b), we have:
  P1 = {10: if ( ), 11:lock *a*, 12: read *x*, 13: unlock *a*}
  P2 = {20:lock *a*, 21:lock *b*, 22:read *y*, 23:write *x*, 24:unlock *b*, 25:unlock *a* }
- A concurrent execution path of a concurrent program is defined to be a sequence of operations executed by all threads, taking into account the global order among threads. Fig. 6 shows four possible examples of concurrent execution paths for the concurrent program in Fig.5.

We define *PATHS* as a set of execution paths *Pi*'s.

PATHS = (P1, P2, P3, …, Pp), where *p* is the number of threads.

Note that PATHS does not take into account the global ordering among threads. For the example in Fig. 6(a) and Fig. 6(b), we have:

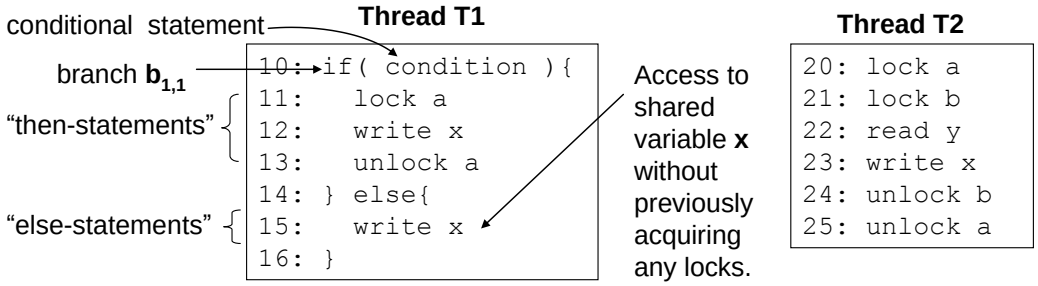


Fig. 5. Example of an if-statement.

PATHS = { *P1*, *P2* } = { {10: if ( ), 11:lock *a*, 12: read *x*, 13: unlock *a*}, {20:lock *a*, 21:lock *b*, 22:read *y*, 23:write *x*, 24: unlock *b*, 25: unlock *a*} }

Interleaving and Branching

We denote by $b_{i,j}$ the *j*-th branch of thread *Ti* in the execution path of thread *Ti*. The truth value of a conditional statement in a branch can be affected by both input values and interleaving because interleaving might affect shared variables, which may in turn affect the conditional statement. Fig. 6(a) and Fig. 6(b) show some possible concurrent execution paths for the program in Fig. 5 when the conditional statement in the branch $b_{1,1}$ is *true*, whereas Fig. 6(c) and Fig. 6(d) show the concurrent execution paths when the conditional statement is *false*.

Let → denotes the "happens-before" relation as follows: If *a* is an event in process *Pi*, and *b* is an event in process *Pj*, then event *a* →event *b* if and only if event *a* happens before event *b*. In the example of Fig. 6, the order of *T1*:10 and *T2*:23 affects the truth value of the branch $b_{1,1}$. The branch is *true* in executions 1 and 2 when *T1*:10 →*T2*:23, and *false* in executions 3 and 4 when *T2*:23 →*T1*:10. We will later explain how to identify operations that affect a branch.

Race Condition

Consistent locking for accessing a shared variable means there is at least one lock which is always acquired by all threads before accessing this shared variable. Such locks are called consistent locks. An access to a shared variable is said to be well formed if all threads acquire a consistent lock before accessing the shared variable, and then perform an unlock operation to release the corresponding lock. In concurrency control using a lock mechanism, a race condition exists when access to a shared variable is not well formed. Detecting race conditions is checking consistent locking for accessing shared variables. A race detector called Eraser [1] proposes an efficient algorithm for checking consistent locking in the execution of a concurrent program. In concurrency control using a lock mechanism, it is the responsibility of programmers that a proper lock operation is performed before accessing a shared variable, and that the lock is released after the access to the shared variable has been completed. There are various reasons why access to a shared variable may not be well formed: for example, programmers may forget to write the lock, they may write an incorrect lock, or they may make an incorrect prediction about the execution path, resulting in the lock not being properly set. An example is shown in Fig. 6(c) and Fig. 6(d) where the "else-statements" in line 15 for thread *T1* access the shared variable *x* without acquiring any locks. Another reason that an access may not be well formed is that programmers may intentionally omit a lock for performance reasons when data race are acceptable, for example by using a volatile variable in Java. In those cases, the access to shared variables is not well formed and a race condition is caused.

"Access-Manner"

We divide an execution path of a single thread into several parts called "access-manners". Later we will show that "access-manner" is useful to define "equivalency" in terms of race condition among different executions of a concurrent program, even though they do not have the same exact sequence of locks and accesses to shared variables. In order to define "access-manner", we use notation *L(Ti)* as the number of active locks acquired by thread *Ti* at a particular time. At the beginning of the execution of thread *Ti*, *L(Ti)* is 0. During an execution of a program, *L(Ti)* is incremented and decremented by the following rules:

- Incremented by 1 when a thread successfully acquires a lock (i.e. has completed a lock instruction).
- Decremented by 1 when a thread releases the lock (i.e. has completed an unlock instruction) that is currently acquired. *L(Ti)* is not decremented if a thread is trying to release a lock that is not currently acquired. Hence *L(Ti)* cannot be negative.

Most race conditions occur because programmers use an incorrect lock or even forget to acquire a lock before accessing shared variables. For checking whether the usage of a lock was correct, we use the term "access-manner" when the access to a shared variable is performed under a lock. We define an "access-manner" as a sequence of operations in which a thread has acquired a lock, has accessed a shared variable, and has released the corresponding lock. An individual "access-manner" is usually a sequence of lock-unlock and read-write operations to shared variables within a thread's execution path that start and end with the following conditions:

- Start: lock operation that causes *L(Ti)* to become 1.
- End: unlock operation that causes *L(Ti)* to become 0, or when execution trace terminates.

An individual "access-manner" must end before another individual "access-manner" starts and there must not be overlaps between "access-manners". In the case where programmers forget to acquire a lock, it is known as an unusual "access-manner" when an access to shared variables without acquiring a lock starts, or when programmers only write an unlock without previously acquiring the lock.

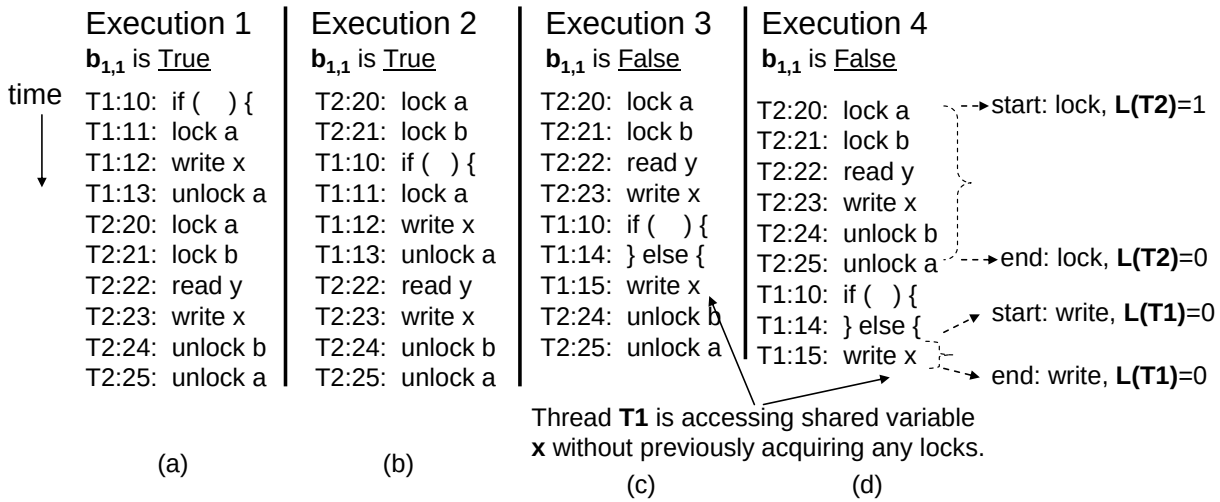


Fig. 6. Example of different concurrent execution paths for program in Fig. 5.

In such a case, "access-manner" is defined as one operation of unlock, read or write to a shared variable. Such an unusual "access-manner" might potentially cause a race condition if another thread is accessing the same shared variable.

Two individual "access-manners" are the same if they have the same sequence of lock-unlock statements and read-write operations on shared variables. We define *Mi* to be a set of "access-manners" for the execution path of thread *Ti*, that is a collection of distinct individual "access-manners" without considering their order. We also define a concurrent set of "access-manners" *MANNERS* = {*M1*, *M2*, *M3*, ... , *MN*} as a collection of sets of "access-manners" from all the threads within a concurrent execution path of a concurrent program. When two concurrent execution paths of a concurrent program have the same *MANNERS*, each thread will have the same set of "access-manners".

When two different concurrent execution paths of a concurrent program have the same *PATHS*, each thread in the two execution paths will exercise exactly the same sequence of lock-unlock and read-write operations on shared variables, hence they will also have the same set of "access-manners". Therefore, two concurrent execution paths with the same *PATHS* will certainly have the same *MANNERS*. The concurrent execution path in Fig. 6(a) and the execution path in Fig. 6(b) have the same *PATHS*, hence they will also have the same *MANNERS*:

M1 = {(11:lock *a*, 12:write *x*, 13:unlock *a*) }

M2 = {(20:lock *a*, 21:lock *b*, 22:read *y*, 23:write *x*, 24:unlock *b*, 25:unlock *a*) }

MANNERS = { {(11:lock *a*, 12:write *x*, 13:unlock *a*)}, {(20:lock *a*, 21:lock *b*, 22:read *y*, 23:write *x*, 24:unlock *b*, 25:unlock *a*)} }

"Race-Equivalent"

Regarding reproducing race conditions due to inconsistent locking for accesses to shared variables, it is beneficial to consider equivalency between two executions of a concurrent program. For this purpose, we introduce a new term called "race-equivalence". Two executions of a concurrent program are "race-equivalent" if they have the same *MANNERS*. "Race-equivalent" means the two concurrent execution paths of a concurrent program have the same consistent locking for accessing shared variables. Different concurrent execution paths of a concurrent program that are "race-equivalent" are said to be in the same "race-equivalent" group. It is sufficient to test only one member from each "race-equivalent" group, thereby reducing the number of interleavings to be tested. For detecting race conditions, we need to check all "race-equivalent" groups.

As explained above, two concurrent execution paths with the same *PATHS* will certainly have the same *MANNERS*. Therefore, two concurrent execution paths of a concurrent program that have the same *PATHS* will certainly be "race-equivalent". As shown before, the execution path in Fig. 6(a) and the concurrent execution path in Fig 6.(b) have the same *PATHS*, so they are "race-equivalent". We can see that lock *a* is a consistent lock for accessing shared variable *x* in both concurrent execution paths. Different "race-equivalent" groups can be created by taking a different concurrent execution path in which at least one thread changes its individual "access-manner". A branch might lead to a different concurrent execution path which, in turn, can produce different individual "access-manners" that can affect consistent locking. As shown in the concurrent execution paths in Fig. 6(c) and Fig. 6(d), there is a race condition because there is no consistent lock for access to shared variable *x* in thread *T1*:

M1 = {(15:write *x*) }

M2 = {(20:lock *a*, 21:lock *b*, 22:read *y*, 23:write *x*, 24:unlock *b*, 25:unlock *a*) }

MANNERS = {{(15:write *x*) }**,** {(20:lock *a*, 21:lock *b*, 22:read *y*, 23:write *x*, 24:unlock *b*, 25:unlock *a*) } }

To detect this race condition, we need only check the concurrent execution path in Fig. 6(c) or the one in Fig. 6(d) because they are "race-equivalent". The same inconsistent locking can be detected.

When a branch changes the execution path of a thread, it might not necessarily produce different consistent locking. In this situation, the same thread in the two concurrent execution paths might not exercise exactly the same lock-unlock sequence and read-write operations on shared

variables, but they will still have the same *MANNERS*, and so we can also classify them as "race-equivalent". This is particularly useful in the case of loops because we do not need to test all the iterations. It is sufficient to test only a partial execution trace from several iterations for checking race conditions because the execution of loop iterations can have the same "access-manners".

In Fig. 7, thread *T1* in execution 1 and execution 2 has different "access-manners", hence concurrent execution paths 1 and 2 are not "race-equivalent". When there is an active lock that was acquired outside the loop, then the first iteration will have different "access-manners" from those in the second iteration because they start from different active locks, as shown in concurrent execution paths 1 and 2 in Fig. 7. On the other hand, concurrent execution paths 2 and 3 in Fig. 7 are "race-equivalent" because each thread in the two executions has the same *MANNERS*:

M1 = { (1:lock *a*, 3:write *x*, 4:unlock *a*), (3:write *x*), (4:unlock *a*) }

M2 = { (20:lock *a*, 21:read *x*, 22:unlock *a*) }

MANNERS = {{(1:lock *a*, 3:write *x*, 4:unlock *a*), (3:write *x*), (4:unlock *a*)}, {(20:lock *a*, 21:read *x*, 22:unlock *a*)}}

The second iteration for the loop accesses the shared variable *x* without previously acquiring any lock, a fact that can be detected in either concurrent execution path 2 or 3. When there is no active lock at the end of a loop, the rest of the iterations will have the same set of "access-manners". The rest of these iterations are called "equivalent iterations" in terms of consistent locking because they have the same set of "access-manners".

Further to the discussion above, the problem for detecting a race condition can be stated as follows:

Given a concurrent program that has an "execution-variant" V_{error} containing an error in its concurrent set of "access-manners" $MANNERS_{error}$, find the V_{error}, or another "execution-variant" V, which has the same concurrent set of "access-manners" as $MANNERS_{error}$. Since each thread in V and V_{error} will have the same set of "access-manners", then the same inconsistent locking and improper lock-unlock sequences in V_{error} will also be detected in V.

B. Reduction in the Number of Different Interleavings

The number of different interleavings is reduced by trying to create only interleavings that lead to a different "race-equivalent" group. This subsection explains how to create different "race-equivalent" groups efficiently. The basic idea is that, for exploring possible different concurrent execution paths caused by branches, it is sufficient to create and test only those interleavings that might affect the conditional statements of branches. Different "execution-variants" from a particular branch *b* might lead to the same value for the condition. Hence, in exploring different concurrent execution paths caused by the branch *b*, we can reduce test cases by grouping those "execution-variants" and testing only one member from each group. We name such a group a "branch-affect" group. "Execution-variants" within the same "branch-affect" group for a branch *b* will have the same condition value for the branch *b*. Let *BranchRelOP(b)* be the set of read and write operations on shared variables that affect the condition of a branch *b*. The idea for grouping the "execution-variants" comes from the fact that if two "execution-variants" execute the same sequence of read and write operations from *BranchRelOP(b)*, then they will give the same condition value for the branch b, and thus they can be grouped into the same "branch-affect" group. Two or more "execution-variants" in the same "branch-affect" group for a branch *b* are redundant with respect to exploring the different concurrent execution paths caused by the branch *b*.

Determining the Set of Operations that Affect Branch Outcomes

In order to identify "branch-affect" groups, we first need to determine the set of operations that affect the conditional value of a particular branch *b*. We propose a data flow analysis method to identify operations that affect the conditional statement of the branch *b* from an execution trace. This method analyzes data flow among accesses to shared variables related to the conditional statement of the branch *b*. Based on this analysis, we can determine which operations are affecting the condition.

One existing method for data flow analysis is by using "use-define". We use "use-define" to find operations that affect conditional statements in branches. First we identify the "use-define" set *SetUD* which we will use for grouping different interleavings. In sequential programs, a "use-define" is a relation consisting of a use, *U*, of a variable, and the definitions, *D*, of that variable that can be reached from that use without any other intervening definitions. A "definition" can have many forms, but is generally taken to mean the assignment operation of some value to a variable. A "use" generally means a read operation on a variable. A "use-define" is a triplet (*variable_name*, *use_location*, *define_location*). R. Caballero, C. Hermanns, and H. Kuchen [23] utilize "use-define" for measuring test coverage but that definition does not apply to concurrent programs. We call the "use-define" for sequential programs the *conventional use-define*. Yang, A.L. Souter, and L.L. Pollock [10] [34] extended the definition and notation of "use-define" to cover possible usages and definitions of shared variables in concurrent programs. They then proposed an automatic generation of concurrent execution paths to cover a particular "use-define" for concurrent programs. In the extended definition of "use-define" for concurrent programs, a use statement includes the usage of a shared variable, and the define statement includes the possibility that the value can be defined from other threads. We call this an *extended use-define*.

Which thread actually defines the value in a particular execution would depend on the interleaving. The "use-define" set can be obtained by analyzing the execution trace or source code. Since the method proposed in this paper iteratively generates different interleavings based on previous execution traces, it is sufficient to use the "use-define" set obtained only from the execution trace. The "use-define" set obtained by the static analysis of source code may contain redundant elements. Information from the source code can be used as a supplement if execution traces

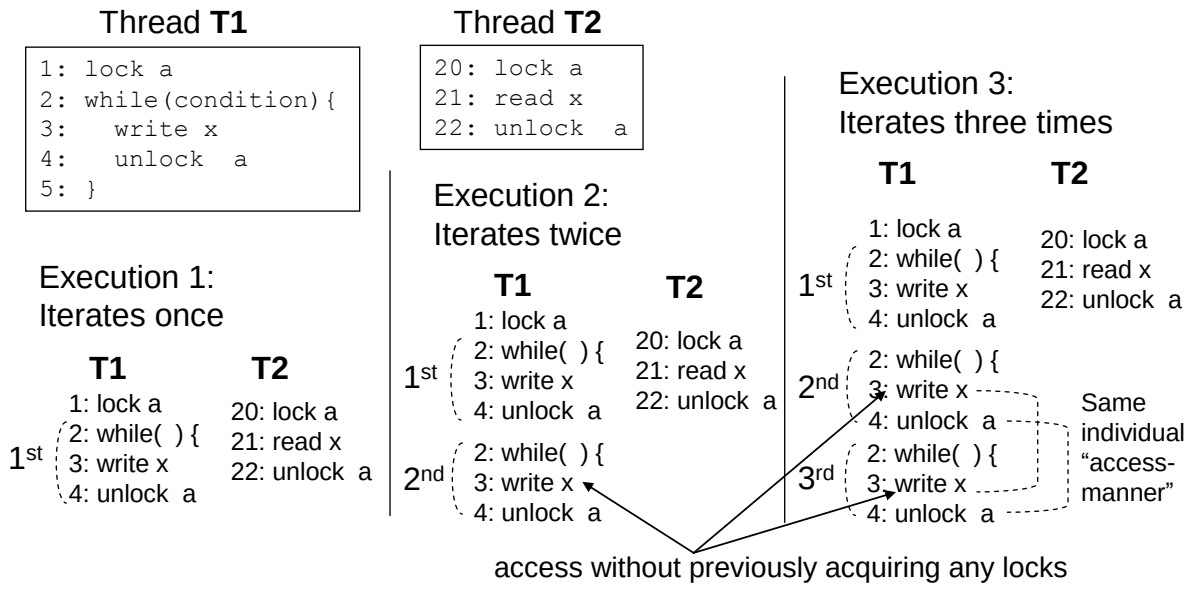


Fig. 7. Example of set of "access-manners" for a loop.

do not contain complete information for obtaining the "use-define" set. In this paper, we assume that the execution trace contains enough information to obtain the "use-define" set consisting of triplets of variable names, read or write operations, and locations. Fig. 8 shows an example of a "use-define" set for the program example in Fig. 3.

To detect a conventional "use-define", we identify the variable in a thread's execution trace and check if it forms a conventional "use-define". To detect an extended "use-define", we first need to identify shared variables from the execution trace. A variable is shared if it is accessed by more than one thread. In the example of Fig. 8, we see that the variable *x* and *y* are shared variables because they were accessed by more than one thread. For each access to a shared variable in a thread, we check if it forms an extended "use-define" with another thread. In the example of Fig. 8, the read operation on shared variable **x** in line 1 and the write operation on shared variable **x** in line 10 form an extended "use-define". There are several examples of "use-define" in Fig. 8, as follows:

- Conventional "use-define": *ud2* = (*n*, 2, 1), *ud4* = (*x*, 12, 10)
- Extended "use-define" for concurrent programs: *ud1* = (*x*, 1, 10), *ud3* = (*x*, 1, 11), *ud5* = (*y*, 7, 11)

Since a wait-notify mechanism can change data flow, it might cause some infeasible "use-defines". This situation could happen, for example, when there is a "wait" command without the corresponding "notify" command. In this example, the "use" or "define" after the wait command will not be executed, so the "use-define" becomes infeasible. C. Yang, A.L. Souter, and L. L. Pollock [10] [34] describe some complications that synchronization causes during data flow analysis. Some infeasible "use-defines" might be included in a "use-define" set, but they will not be executed and will not be used for grouping "execution-variants". The infeasible "use-define" pairs will cause redundancy in the "use-define" set, but they will not cause redundancy in test case generation.

Data Flow Relation with "use-define"

For identifying data flow, we define a dependency relation between "use-defines". A "use-define" *ud2* depends on another "use-define" *ud1*, if the "definition" for the variable in "use-define" *ud2* is using the variable in the "use-define" *ud1*. It is basically a dependency relation. When there can be only one assignment statement for every line of code, a "use-define" *ud2* depends on another "use-define" *ud1* when the following condition is satisfied:

define_location of "use-define" *ud2* = = use_location of "use-define" *ud1*

An example of a dependency relation between "use-defines" is shown in Fig. 8. Since the *def_location* of "use-define" *ud1* is the same as the *use_location* of "use-define" *ud2*, "use-define" *ud2* depends on "use-define" *ud1*. This means that there is data flow from the variable *x* to the variable *n*, because the definition of variable *n* in line 1 uses the variable *x* in line 10. In a similar way, the "use-define" *ud2* depends on the "use-define" *ud3*.

We define *BranchRelUD(b)* as a set of "use-defines" on which a conditional statement of a branch *b* could depend. It is basically a dependency relation. Algorithm 2 shows how to find the members of *BranchRelUD(b)* using the dependency relation of "use-define". We also define *BranchRelOP(b)* as a set of read and write operations from members of *BranchRelUD(b)*. They are read and write operations on shared variables from the "use-define" that has a data flow relation with the variables in the conditional statement of a branch *b*. Members of *BranchRelOP(b)* are operations in threads which are defined as triples:

location:*operation*:*operand*

as defined in the *Model for Concurrent Program Execution Traces* subsection.

Algorithm 2. Finding a set of operations that is affecting branch outcomes.

Input:
- *SetUD*: set of "use-defines" from a concurrent program execution trace.
- A branch *b*.

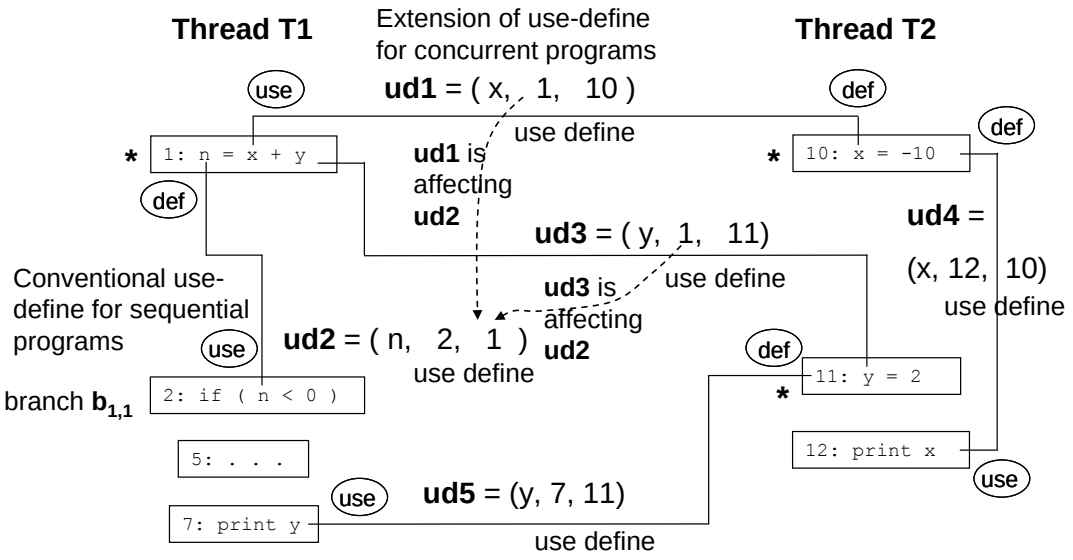


Fig. 8. Examples of "use-defines" for the concurrent program in Fig. 3.

Output:

- *BranchRelOP(b)*: a set of read and write operations that is affecting the conditional statement of branch *b*.

Step 1. Initialization.

1.1 *BranchRelUD(b)*: "use-defines" from *SetUD* where the variables are used in the conditional statement of the branch *b*.

Step 2. Find all related "use-defines".

2.1 **For** each "use-define" *ud* in *SetUD*, where

ud is not included in *BranchRelUD(b)*, and

ud does not contain any operations from the same thread as the branch *b* after the execution of the branch *b*.

2.1.1 Check if *ud* is affecting any "use-define" in *BranchRelUD(b)*.

2.2 **If** any "use-define" is found in **Step 2.1.1**

Then Add all the "use-defines" found in **Step 2.1.1** to *BranchRelUD(b)* and repeat **Step 2.1**.

Or Else Go to **Step 3**.

Step 3. Find read and write operations that are affecting the conditional statement of branch *b*.

3.1 Take all the read and write operations from members of *BranchRelUD(b)*, and add them into *BranchRelOP(b)*.

When Algorithm 2 no longer finds "use-defines" that satisfy the conditions in step 2.1.1, it means that all "use-defines" related to the conditional statement of the branch *b* have been included in *BranchRelUD(b)*. In step 3, *BranchRelOP(b)* contains all read and write operations on shared variables that affect the conditional statement of the branch *b*. Fig. 9 shows an example of a *BranchRelOP(b)*. When we consider different effects caused by interleaving, we need only consider different interleavings of read and write operations on shared variables. Note that *BranchRelOP(b)* is calculated from a particular execution path and a particular execution trace. If a new test case explores a different execution path, then new members for *BranchRelOP(b)* might need to be added. The example in Table I illustrates how Algorithm 2 finds *BranchRelOP*($b_{1,1}$) for the program example in Fig. 8. The *BranchRelOP*($b_{1,1}$) operations are indicated by asterisks (*) in Fig.8.

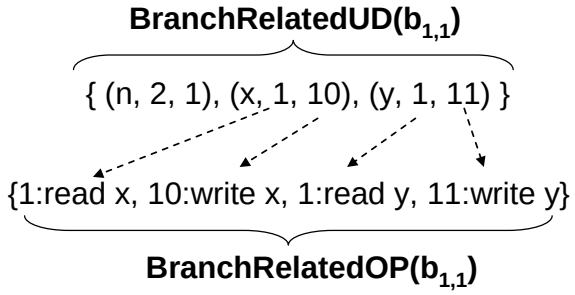


Fig. 9. Obtain the *BranchRelOP*($b_{1,1}$) operations from their *BranchRelatedUD*($b_{1,1}$).

Grouping "Execution-Variants" That Have the Same Effect on Branch Outcomes

Only interleavings that might affect the condition values of branches need be tested for exploring possible different concurrent execution paths created by the branches. We define Algorithm 3 for creating "branch-affect" groups by grouping "execution-variants" that give the same condition value for a branch *b*.

Algorithm 3. Creating a set of "branch-affect" groups for a branch.

Input: "execution-variants" from a variant graph

Output: A set of "branch-affect" groups *G(b)* for a branch *b*.

G(b) = {*g1(b), g2(b), g3(b), ...* }, where *g1(b), g2(b), g3(b)* are the first, second, and third "branch-affect" groups for the branch *b* in the execution trace.

Step 1. Find *BranchRelOP(b)* using Algorithm 2.

Step 2. For each "execution-variant" *V* in the variant graph.

2.1 **If** the order of operations from *BranchRelOP(b)* in the current "execution-variant" already exists in the "branch-affect" groups.

Then Add the current "execution-variant" into the corresponding existing "branch-affect group".

Else Add a new "branch-affect" group into *G(b)*

Include the current "execution-variant" *V* in the new "branch-affect" group.

As shown in the example in Fig. 10, "execution-variants" *V3* and *V4* can be grouped together into the same

"branch-affect" group with respect to the branch $b_{1,1}$ because they execute the same order of operations from *BranchRelOP*($b_{1,1}$). A similar situation also applies for the "execution-variants" *V6* and *V7*, as shown in Fig. 10. Table II shows the complete groups for the examples in Fig 10.

As mentioned in Section IV.A, two different concurrent execution paths with the same set of execution paths *PATHS* will be "race-equivalent". To explore different "race-equivalent" groups, it is necessary to find different sets of execution paths *PATHS*. Since the execution path of a thread is affected by branches, we introduce a "branch-condition" table to measure the progress of a test. A "branch-condition" table contains a list of all possible sets of execution paths *PATHS*. Each row in a "branch-combination" table represents the condition values of if-statements and the number of iterations for loops in a concurrent execution path, so each row represents a possible set of execution paths *PATHS*. Each different loop iteration will lead to a different execution path, so we need to consider all loop iterations. However, if loop iterations have the same

TABLE II
EXAMPLE OF A "BRANCH-AFFECT" TABLE.

Branch	Members of "branch-affect" groups	Order of operations from *BranchRelOP*($b_{1,1}$)
$b_{1,1}$	$g1(b_{1,1}) = \{V1\}$	1:read x → 1:read y
	$g2(b_{1,1}) = \{V2\}$	1:read x → 10:write x → 1:read y
	$g3(b_{1,1}) = \{V3, V4\}$	1:read x → 10:write x → 11:write y → 1:read y
	$g4(b_{1,1}) = \{V5\}$	10:write x → 1:read x
	$g5(b_{1,1}) = \{V6, V7\}$	10:write x → 11:write y → 1:read x

set of "access-manners", then there is no need to check all of the iterations because they will be "race-equivalent". A "branch-combination" table is an accumulation from each execution of a test case. It is possible that not all branches can be identified from the execution trace of the first test case. If new branches are found during the execution of the next test case, they should be added to the "branch-combination" table. At the beginning, all rows are marked as "untested", except for the one corresponding to the execution in the first test case.

An example of a "branch-condition" table is shown in Fig.11. We need to test all the feasible sets of execution paths *PATHS*; that is, in order to find the inconsistent locking for accesses to shared variables that have caused errors, all the rows in a "branch-combination" table need to be tested. Algorithm 4 is the complete algorithm of the proposed method. This algorithm integrates the existing reachability testing in step 1.2, with the deterministic testing and race detection in step 4.

Algorithm 4. Complete algorithm for generating test cases and testing consistent locking

Definitions:

- *Outcome*($gk(b_{i,j})$) is the truth value for an if-statement or the number of iterations for a loop of a "branch-affect" group $gk(b_{i,j})$

- *Outcome*(r, $b_{i,j}$) is the truth value or the number of iterations of the branch $b_{i,j}$ for row r in a "branch-condition" table.

Input: a concurrent program and its input.

Output: test cases and race-detection results.

Step 1. Initialization:

1.1. Re-execute the concurrent program taking tracing using the same input as when the error occurred.

1.2. Create the corresponding variant graph from the execution trace using Algorithm 1.

1.3. Create a "branch-condition" table based on the execution trace from step 1.1.

1.4. For each branch of the variant graph in step 1.3, classify each "execution-variant" into "branch-affect" groups using Algorithm 3.

Step 2. Conditions for termination.

2.1 Terminate this algorithm if at least one of the following conditions is satisfied:

- Condition 1: all rows in the "branch-condition" table have been tested,

- Condition 2: all "branch-affect" groups have been marked as "tested". Note that the algorithm terminates with the second condition if there exists any infeasible set of concurrent execution paths for the given input.

Step 3. Select the next test cases *TestCases*:

3.1 *TestCases* = { Ø }

3.2 **For** each untested row r in "branch-condition" table

3.2.1 *Candidates* = { Ø }, *firstGroup* = *true*.

3.2.2 **For** each branch $b_{i,j}$.

If (*firstGroup* = = *true*).

Then *Candidates* = all members of "branch-affect" groups of branch $b_{i,j}$ where *Outcome*($gk(b_{i,j})$) == *Outcome*(r, $b_{i,j}$)

firstGroup = *false*

Else *Candidates* = *Candidates* ∩ all members of the "branch-affect" groups of the branch $b_{i,j}$ where *Outcome*($gk(b_{i,j})$) == *Outcome*(r, $b_{i,j}$)

3.2.3 Select one "execution-variant" from *Candidates* and add it to *TestCases*.

3.2.4 **If step 3.2.3** does not produce any test cases.

Then choose a member from an untested "branch-affect" group and add it to the *TestCases*.

Step 4. Test cases execution.

4.1 Execute the "execution-variants" from the *TestCases* using deterministic testing with tracing.

4.2 Check the execution trace from **step 4.1** using an existing race detector and report any errors.

4.3 Derive new "execution-variants" from the execution trace in **step 4.1**, update the variant graph and "branch-condition" table.

4.4 Classify the new "execution-variants" into "branch-affect" groups.

Step 5. Repeat from **step 2**.

"Race-equivalent" means two concurrent execution paths of a concurrent program have the same consistent locking for accessing shared variables, and also share the same proper/improper lock-unlock sequences. When a variant graph produces "execution-variants", our algorithm groups them into "race-equivalent" groups. Our method achieves test case reduction by testing only one member of each

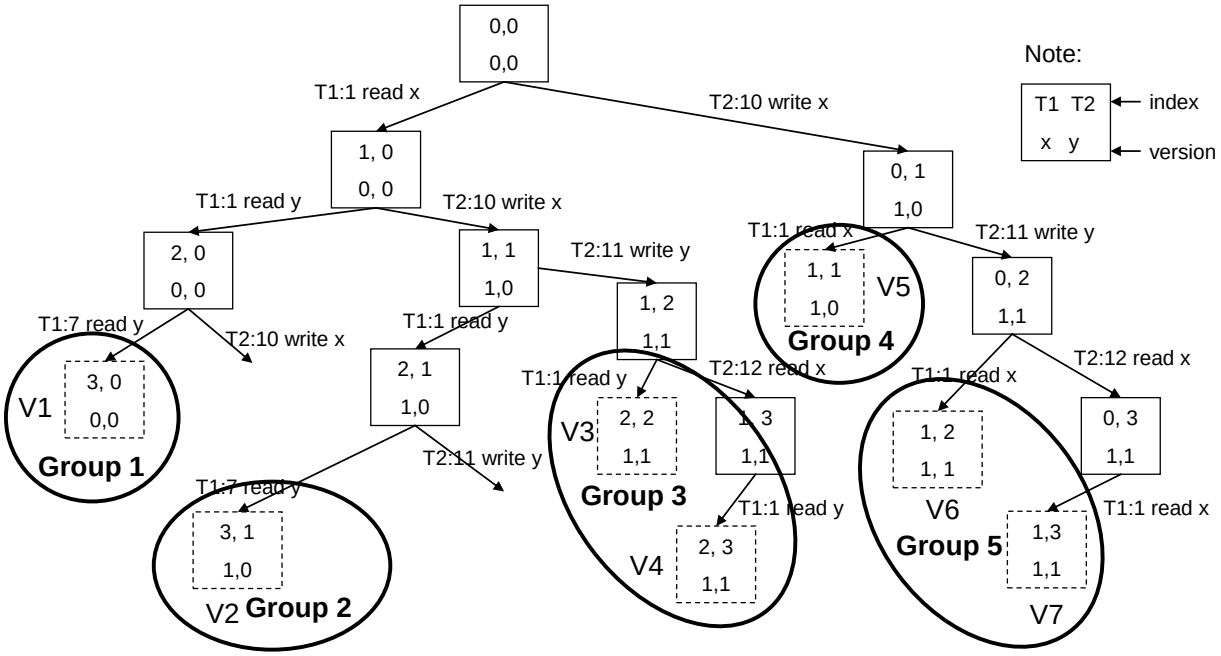


Fig. 10. Examples of "branch-affect" groups for the variant graph in Figure 4.

"race-equivalent" group.

Step-by-step Example of Algorithm 4

A step-by–step example of Algorithm 4 is shown in Tables III and IV. We assume that there is a concurrent program with two threads *T1* and *T2*. Thread *T1* has one branch $b_{1,1}$ and thread *T2* has one branch $b_{2,1}$. The branches $b_{1,1}$ and $b_{2,1}$ are if-statements The steps in Table III are deduced from the analysis shown in Fig. 11. The steps in Table IV are deduced from the analysis shown in Fig. 12.

TABLE III
STEP-BY-STEP EXAMPLE OF ALGORITHM 4.

Step	Description
1	Let us assume that step 1 results a variant graph with five "execution-variants". The execution for the first test case is *V1* which makes $b_{1,1}$ and $b_{2,1}$ *True*. Assume that the "branch-affect" group has been calculated using Algorithm 3 and the "branch-condition" table is as exemplified in Fig. 11.
2	Not all rows in the branch-condition table have been tested, so proceed to Step 3.
3	Step 3.2.3 does not produce any test cases.
3.2.4	Since Step 3.2.3 does not find any test cases, *V5* is chosen as a test case from untested branch affect group $g2(b_{1,1})$.
4.1	Execute *V5* using deterministic testing and obtain execution trace.
4.3	When we derive the execution trace from step 4.1, we find new "execution-variant" *V6*
4.4	The new "execution-variant" *V6* is classified into $g2(b_{1,1})$ and $g1(b_{2,1})$, see Fig. 12.
5	Repeat from step 2

There is no need to test the "branch-affect" group $g3(b_{1,1})$ because all the rows in the "branch-condition" table in Fig. 12 have been completed. Our algorithm only requires the testing of four "execution-variants" from the total of six "execution-variants".

C. Extension of a Variant Graph for Handling Synchronization

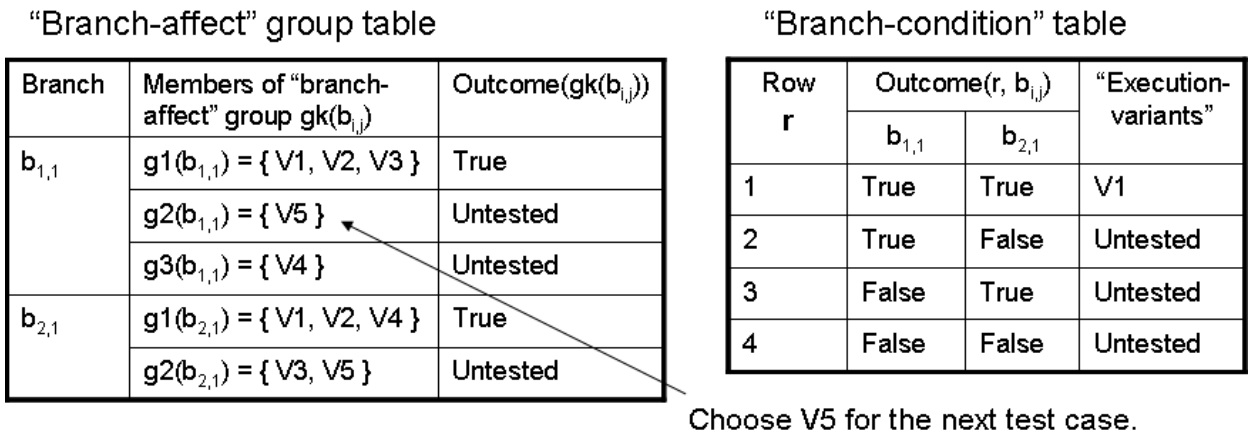
We extend the existing variant graph [11] by considering synchronization dependencies to eliminate redundancy and

TABLE IV
STEP-BY-STEP EXAMPLE OF ALGORITHM 4 (CONTINUED).

Step	Description
2	Not all rows in the branch-condition table have been tested, so proceed to Step 3.
3	*TestCases* = { Ø }, for each untested row *r* in the "branch-condition" table The 2nd row: *Candidates* = {*V1*, *V2*, *V3* } ∩ {*V3*, *V5*} = *V3*. The 3rd row: *Candidates* = {*V5*, *V6*} ∩ {*V1*, *V2*, *V4*, *V6*} = *V6*. The 4th row: *Candidates* = {*V5*, *V6*} ∩ {*V3*, *V5*} = *V5*. *TestCases* = { *V3*, *V6*, *V5*}
4	No need to do step 4 because there are some test cases from step 3.
5.1	Execute the members of *TestCases*.
5.2	No new "execution-variants" can be derived from the trace in step 5.1.
2	All rows in the "branch-condition" table have been tested, so the algorithm terminates.

to avoid false negatives. The extended model for variant graphs utilizes trace information about lock-unlock and wait-notify operations. For wait-notify, we assume a simple model in which a thread that is waiting can receive a notification from any thread and a notification is sent to all threads. Only waiting threads can accept the notification, otherwise the incoming notification will be lost. We extend the node in a variant graph to include flags for "lock" and "wait" besides the existing "index" and "version". "Index" will also be incremented for lock-unlock and wait-notify operations. In this way, different orders of wait-notify will be considered in test case generation, thus avoiding false negatives. We add the following rules in the extended variant graph for handling lock-unlock and wait-notify operations:

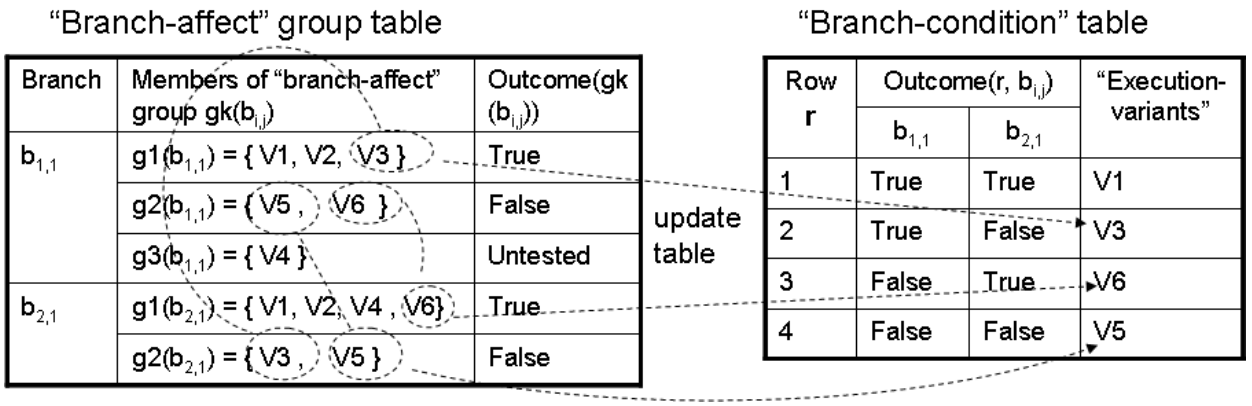
- Lock-unlock:
  - If the operation is "lock", set the lock flag for the corresponding lock to 1.
  - If the operation is "unlock", reset the lock flag for the corresponding lock to 0.



Branch	Members of "branch-affect" group $gk(b_{i,j})$	Outcome($gk(b_{i,j})$)
$b_{1,1}$	$g1(b_{1,1})$ = { V1, V2, V3 }	True
	$g2(b_{1,1})$ = { V5 }	Untested
	$g3(b_{1,1})$ = { V4 }	Untested
$b_{2,1}$	$g1(b_{2,1})$ = { V1, V2, V4 }	True
	$g2(b_{2,1})$ = { V3, V5 }	Untested

Row r	Outcome(r, $b_{i,j}$)		"Execution-variants"
	$b_{1,1}$	$b_{2,1}$	
1	True	True	V1
2	True	False	Untested
3	False	True	Untested
4	False	False	Untested

Fig. 11. "Branch-affect" group table and "branch-condition" table for the first test case



Branch	Members of "branch-affect" group $gk(b_{i,j})$	Outcome(gk ($b_{i,j}$))
$b_{1,1}$	$g1(b_{1,1})$ = { V1, V2, V3 }	True
	$g2(b_{1,1})$ = { V5 , V6 }	False
	$g3(b_{1,1})$ = { V4 }	Untested
$b_{2,1}$	$g1(b_{2,1})$ = { V1, V2, V4 , V6}	True
	$g2(b_{2,1})$ = { V3 , V5 }	False

Row r	Outcome(r, $b_{i,j}$)		"Execution-variants"
	$b_{1,1}$	$b_{2,1}$	
1	True	True	V1
2	True	False	V3
3	False	True	V6
4	False	False	V5

Fig. 12. "Branch-affect" group table and "branch-condition" table when Algorithm 4 terminates.

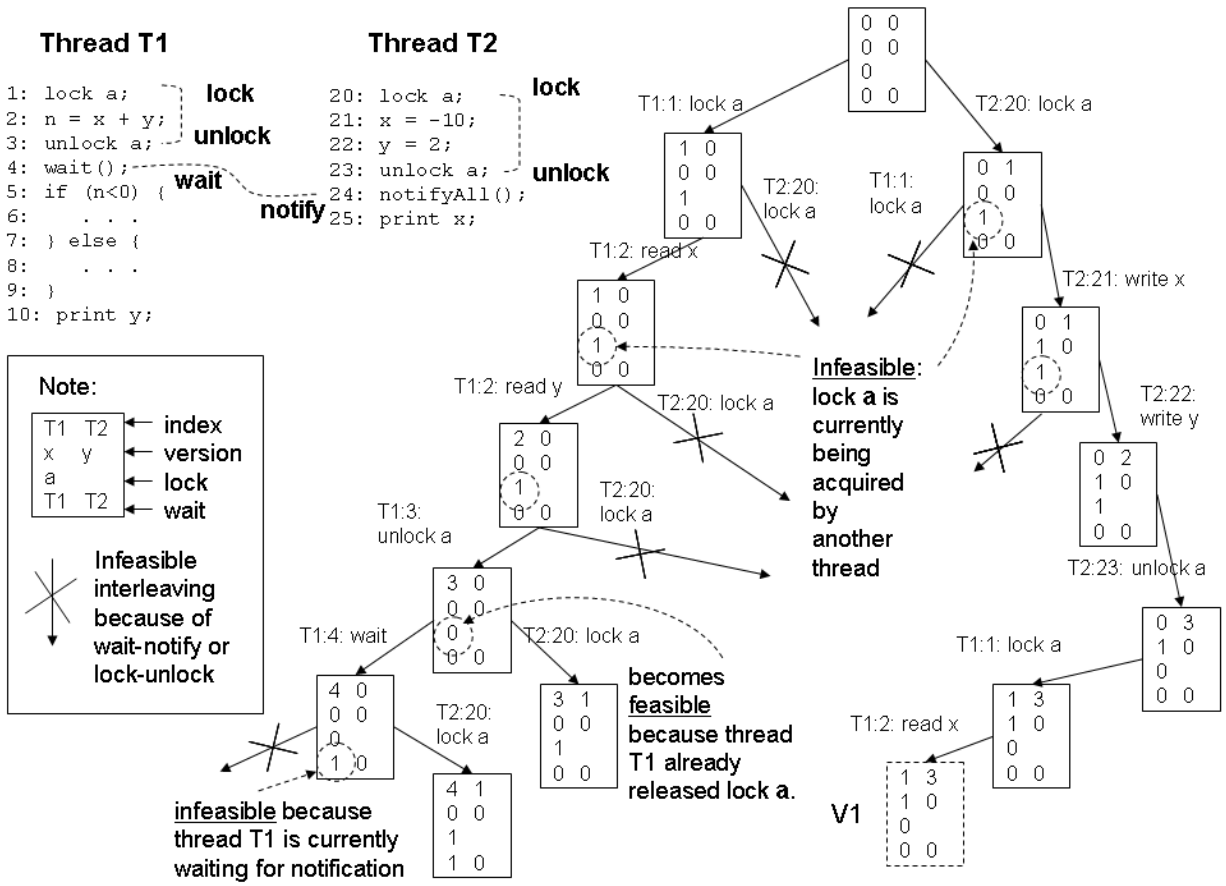


Fig. 13. Example of the extension of a variant graph.

- Wait-notify:
  - If the operation is "wait", set the wait flag for the corresponding thread to 1.
  - If the operation is "notify", reset the wait flags for all threads to 0.

When expanding an extended variant graph, a node is infeasible if any one of the following conditions holds:

- The wait flag for the corresponding thread is 1.
- The operation is lock and the lock flag is 1.

Fig. 13 shows an extension of a variant graph which adds lock-unlock and wait-notify operations for the concurrent program in Fig. 3(a). The extended variant graph in Fig. 13 identifies some infeasible interleavings caused by the lock-unlock and wait-notify operations.

D. Correctness

Theorem 1: Given a particular input, Algorithm 4 guarantees that any error caused by inconsistent locking in an execution of a concurrent program will be detected.

Proof: Let us assume that a concurrent program has an "execution-variant" V_{error} that contains an error with its concurrent set of "access-manners" $MANNERS_{error}$. We need to show that our algorithm will detect an "execution-variant" **V** in the test cases that has the same concurrent set of "access-manners" as $MANNERS_{error}$. Since V_{error} and V are in the same "race-equivalent" group, the same inconsistent locking and improper lock sequence will be reproduced when the algorithm terminates.

There are two conditions for termination in the Algorithm 4 step 2:

The first condition for termination ensures that all possible concurrent execution paths have been tested, so it is certain that one set of concurrent execution paths will be the same as $PATHS_{error}$. Since two concurrent executions with the same set of concurrent execution paths will certainly have the same $MANNERS_{error}$, the same inconsistent locking will be reproduced. In the case of loops, it is ensured that all equivalent iterations in terms of race condition have been tested.

The second condition for termination ensures that all "branch-affect" groups have been tested, but not all possible concurrent execution paths are tested because some might be infeasible. When Algorithm 4 iterates, it chooses an "execution-variant" as a test case from an untested "branch-affect" group in step 4. The same "execution-variant" as V_{error} or an "execution-variant" with the same $PATHS_{error}$ or $MANNERS_{error}$ might be chosen, so the same cause of error will be detected. Otherwise the algorithm keeps iterating from step 2 to step 5. It updates the "branch-affect" groups in step 5.4 until all are created and tested; hence we know the condition value or the number of iterations for all the "branch-affect" groups. The intersection of *Candidates* in step 3.2.2.1.1 will produce a non-empty feasible set of concurrent execution paths. If V_{error} exists, then $PATHS_{error}$ would also certainly be feasible, so the *Candidates* will contain some members. Step 3.2 will select an untested row r in the "branch-condition" table for which the outcomes for each branch will be the same as in V_{error}. For this row r, branches for the members in *Candidates* will have the same condition values for if-statements as in V_{error} and equivalent iterations for loops in terms of "access-manners". Therefore, they will have the same concurrent set of "access-manners" as $MANNERS_{error}$, and thus the same inconsistent locking will be detected. *QED*.

E. False Alarms

False Positives

The occurrence of false positives depends on the precision of the information from the execution trace. In the proposed method, we use an execution trace that contains information about access to shared variables, lock acquisition, and branches, but we currently do not consider indexes of arrays or commands to "wait" for a fixed period of time. We explore some of the possibilities for the occurrence of false positives.

Arrays: when an array is shared, the actual element that is shared depends on the index of the array. The index could be specified by a variable whose value can depend on input and interleaving. Since our method currently does not consider the value of the index variable in test case generation, we cannot guarantee whether two or more threads are always accessing the same or a different element of the array. To be safe, we take a conservative approach by considering all elements in an array to be shared when we are checking for race conditions. Unfortunately, locking all elements in an array would decrease concurrency because other threads have to wait to access different elements. To increase concurrency, sometimes programmers divide the values of the index into several groups and use separate locks for each group consistently during programming. In this situation, our proposed method results in false positives.

Wait: using commands to "wait" for a fixed period of time, for example wait(100ms), will cause some interleavings to become infeasible. Some commands in the same thread after a "wait" command would not be interleaved with other threads because the thread is suspended for a period of time. For example, in an extreme situation, other threads might have finished, so the waiting thread continues its own execution without interleaving with any other threads. Since our current method does not consider the usage of "wait" command for a fixed period of time, our algorithm might generate some interleavings that are infeasible. However, we consider using a command to "wait" for a fixed period of time to be a bad programming practice.

False Negatives

The basic premise suggested in this method is that covering an execution path is sufficient to detect a race or no-race condition by checking consistent locking in that concurrent execution path, independent of variable values. In some cases, this might not be sufficient, since the value of the lock object itself may depend on the data flow and, theoretically, on the interleaving. Similar problems may also arise when different shared reference variables (a pointer in C or an object reference in Java) actually refer to the same data. Threads acquire a consistent lock for accessing different reference variables, but actually they are referring to the same data. On the other hand, even when the same reference variable is shared between threads, the actual data referred to may not necessarily be shared. This situation may be considered to be a race condition, but currently it is

TABLE V
SUMMARY OF EXPERIMENTAL RESULTS

	Apache Commons Pool	Jtelnet	jNetMap	JoBo	Apache Derby
Program size (Kloc)	123	5	3	45	292
Trace size (KB)	35	1638	201	87500	72800
Number of threads	3	3	6	4	5
Number of shared variables	33	7	10	4	33
Number of branches executed from trace	17	329	31	121665	14164
Number of branches affected by interleaving	1	0	1	1	29
Number of test cases in TPAIR	23	66	Infinite	Infinite	1453539
Number of test cases in proposed method	2	0	5	2	58

beyond the scope of our proposed method, so our algorithm might produce false negatives. However, such a situation is rare in programming practice and is not recommended because it can confuse programmers about the usage of object locks.

V. EXPERIMENTS

A. Lock Mechanism and Tracing in Java

Lock Mechanism in Java

In Java, the lock mechanism is implemented as follows:

1. Lock object: an actual object that represents a lock. One example of an implementation class is ReentrantLock. A lock is acquired by calling the *lock()* method and released by calling the *unlock()* method.
2. Synchronized method: a method which has a "synchronized" keyword in its method declaration. A synchronized method uses the object instance (i.e., this) as the lock object, unless it is a static method, in which case the lock is the class lock. A thread that wants to execute a synchronized method must first obtain the lock. The lock is released after it returns from the synchronized method.
3. Synchronized statement: similar to synchronized method, but an object that provides the lock must be specified.

For the three mechanisms above, a lock is being acquired irrespective of which syntactic approach is used.

Tracing

We use AspectJ [25] for tracing Java multi-threaded concurrent programs. AspectJ was chosen because of its ability to trace the necessary data from an execution of a program. Other means of tracing can also be used as long as they can capture the necessary information about lock sequences, access to shared variables, and branches. We capture the necessary information from an execution of a program using the concept of "pointcut", "advice", and "reflection" in AspectJ. Note that "pointcut", "advice", and "reflection" are specific terms for AspectJ. Here, we only describe the general idea of tracing using AspectJ.

We wrote "pointcut" to specify locations within an execution of a program where necessary information needs to be captured. We do not explicitly specify the locations; instead we specify wildcards so AspectJ will take a trace when any locks are acquired or released, or any shared variables are accessed. For each lock acquisition and access to a shared variable, we write a corresponding additional piece of code to be executed, called "advice". Within the "advice", we use "reflection" to get the necessary information for tracing, such as the shared variable's name. For detecting branches, the line of code in the source code is recorded when a variable is accessed, and then later compared to the source code to determine whether it is in an if-statement or a loop. The AspectJ codes necessary for tracing are written in AspectJ files, which are separated from the target programs. The target source code must then be weaved with AspectJ codes.

B. Experiment Results

We use some Java open source programs in network control and database management for the experiments, because these programs are usually designed to be multithread. The objective of the experiments is to show the effectiveness of the proposed method for reducing the number of test cases in detecting race conditions. Later we will compare the number of test cases against an existing test case reduction method based on the Thread-Pair-Interleaving (TPAIR) criterion [47]. The results are summarized in Table V. For a fair comparison, we allow only the same input for both methods. In these experiments, we measure the reduction in the number of different interleavings used for test case generation. We ignore different orders of read-shared variables. A read-shared variable is a variable that it is written during initialization only and becomes read-only thereafter [1]. Its value is determined only by the input and it does not change during an execution of a program. As such, it can be ignored during test case generation because different interleavings do not affect its value.

Experiment 1: Apache Commons Pool [46]

In Experiment 1, we use a generic object-pooling library called Apache Commons Pool. Some race conditions have been reported in related work [26] [48]. Most of the race conditions are easy to detect in that they can be found by simply re-executing the program and using an existing dynamic race detector. Our proposed method is intended to find race conditions that are difficult to detect. This is because such race conditions are affected by branches and different interleavings. There are 160 race conditions reported at [26]. We observed 15% of them as being difficult to detect. One possible example is shown in Fig. 14.

There is a race condition in Fig. 14 between thread *T1* and thread *T2* when accessing the shared variable *_factory*,

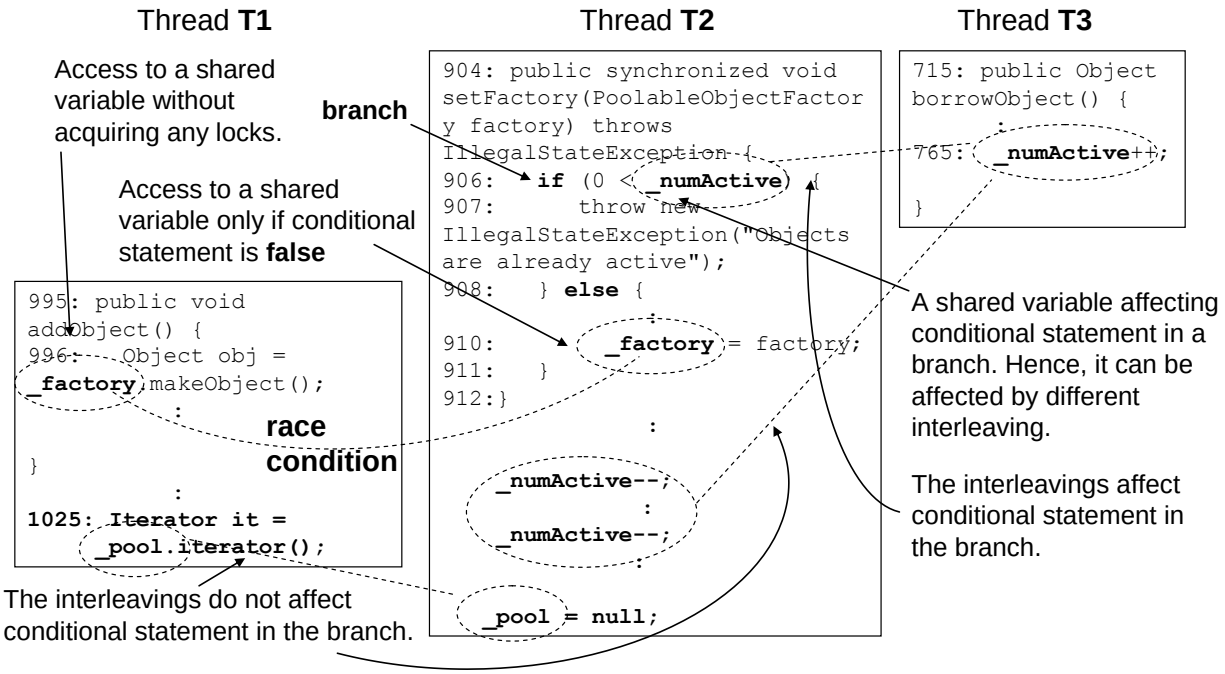


Fig. 14. Example of a race condition that is difficult to detect.

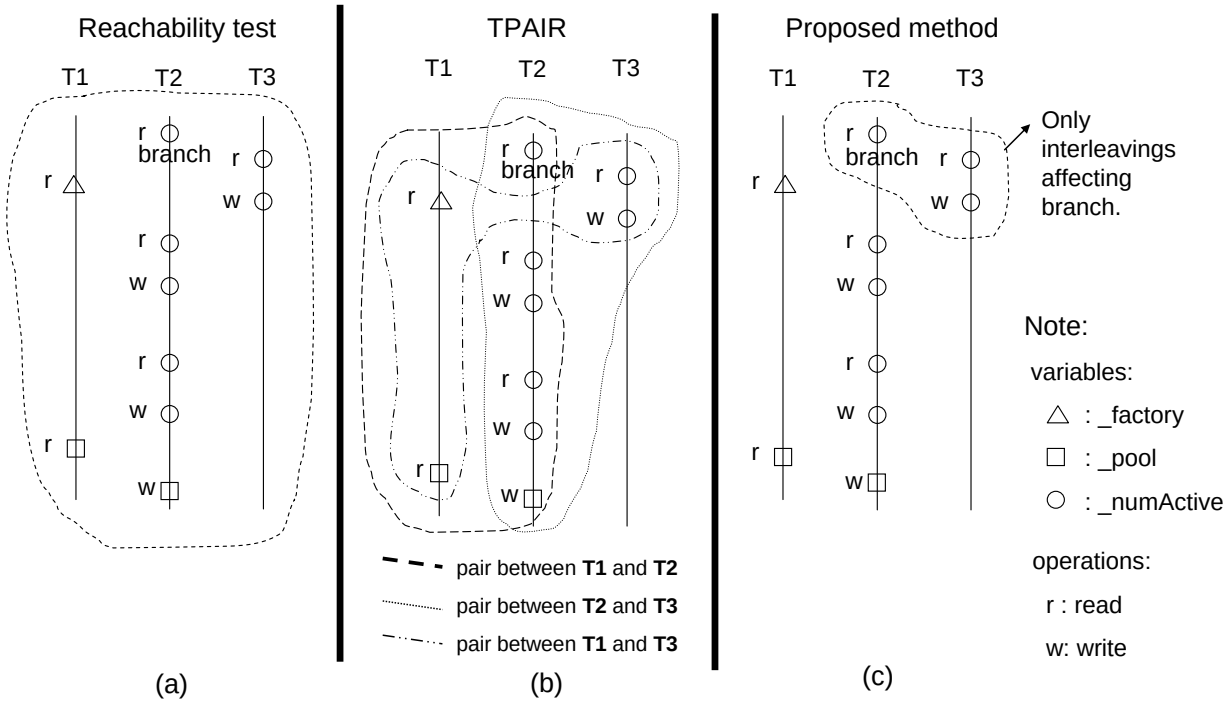


Fig. 15. A comparison of reachability testing, TPAIR, and the proposed method.

because thread *T1* does not acquire any locks. However, it happens only when the conditional statement for the branch in thread *T2* is *false*. Furthermore, the conditional statement depends on the value of shared variable *_numActive* which is affected by interleaving with thread *T3*. Fig. 15 shows read and write accesses to the shared variables for the execution of the first test case, in which the race condition is not reproduced. Using Algorithm 2, we calculate the following:

BranchRelUD(b) = { (*_numActive*, 906, 765) }.

BranchRelOP(b) = { 906: read *_numActive*, 765: write *_numActive* }

Our proposed method generates two test cases based on Table VI. Group *g2*(*b*) will cause the conditional statement to become *false*, so the error will be reproduced.

TABLE VI
TEST CASES FOR EXPERIMENT 1.

Groups	Order of operations from *BranchRelOP(b)*
g1(b)	765: write *_numActive* → 906: read *_numActive*
g2(b)	906: read *_numActive* → 765: write *_numActive*

We compare our proposed method against an existing test case reduction method based on the Thread-Pair-Interleaving (TPAIR) criteria [47]. Instead of generating different interleavings among all threads, TPAIR only generates different interleavings for every pair of threads to reduce the number of test cases. This reduction is based on the fact that most concurrency bugs are caused by the interaction between

TABLE VII
"BRANCH- AFFECT" GROUPS FOR JNETMAP

"Branch- affect" groups	Order of operations from *BranchRelOP(b)*
$g1(b_{2,1})$	*T2*:279: read *pingInterval* → *T-AWT*:112: write *pingInterval*
$g2(b_{2,1})$	*T-AWT*:112: write *pingInterval*, *T2*:279: read *pingInterval*
$g1(b_{2,2})$	*T2*:286: write *pingInterval* → *T2*:279: read *pingInterval* → *T-AWT*:112: write *pingInterval*
$g2(b_{2,2})$	*T2*:286: write *pingInterval* → *T-AWT*:112: write *pingInterval* → *T2*:279: read *pingInterval*
$g3(b_{2,2})$	*T-AWT*:112: write *pingInterval* → *T2*:286: write *pingInterval* → *T2*:279: read *pingInterval*

two threads, instead of all threads, as explained in the previous error detection work [1] [24]. This also happens for the race condition between thread *T1* and thread *T2* when accessing shared variable *_factory* in Fig. 14. Its reproduction depends on the branch in thread *T2* whose conditional statement is affected by the interleaving between thread *T2* and thread *T3*. However, not all different interleavings between those two threads will affect the reproduction of the race condition. For example, shared variable *_pool* is affected by the interleaving between thread *T1* and thread *T2*, but the race condition when accessing the shared variable *_pool* will always be reproduced. Hence, it can always be detected by a race detector independent of the interleaving between those two threads. In this experiment, the reachability testing method produces 147 test cases, the TPAIR method produces 23 test cases, and our proposed method produces only 2 test cases for detecting the race condition.

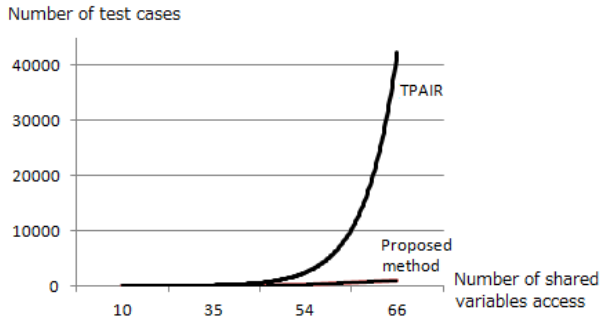


Fig. 16. Comparison of numbers of test cases.

In order to evaluate the feasibility, we performed several experiments by increasing the number of shared variables accesses for the same target program. Fig. 16 indicates the increase in the number of test cases when the number of accesses to shared variables is increased. In order to reproduce the race condition, Fig. 16 shows that our proposed method produces fewer test cases than test generation based on the existing TPAIR. In addition, error detection by TPAIR can be guaranteed only if the errors are caused by interteaving between two threads. In contrast, our proposed method can reproduce errors caused by interleaving from any number of threads, precisely because it takes into consideration data flow that affects the conditional statement in the branch.

Experiment 2: JTelnet [29]

The JTelnet is a telnet client written in Java. Among the 7 shared variables, 6 of them are read-shared. Based on the data flow analysis, one branch is affected by a shared variable. This experiment shows that some interleaving will change the values of shared variables, but they might not affect the reproduction of race conditions. In such circumstances, the existing reachability testing and TPAIR methods will generate test cases, while our proposed method generates no test case. The results are summarized as follows:

- TPAIR (66 test cases): Test cases generated by TPAIR will affect only the values of shared variables in thread AWT-EventQueue-0, but will not affect any conditional statements for branches in thread *T2* (Fig. 17).
- Our proposed method (0 test cases): Branches in thread *T2* are only affected by operations in the same thread. Therefore, the proposed method does not produce any test cases because their outcomes will not be affected by a different interleaving.

Experiment 3: jNetMap [28]

The jNetMap program is a network client to monitor devices in a network. This program detects PCs and a router in a network. Among the 10 shared variables, 9 of them are read-shared variables. Based on data flow analysis, the one non read-shared variable affects one branch. The source code and its execution trace are shown in Fig. 18 and Fig. 19. The results are summarized as follows:

- TPAIR (infinite test cases): There is an infinite loop affecting the read and write sequence which causes infinite test case generation because it considers different values of shared variables as different test cases.
- Our proposed method (5 test cases): There are two test cases from the "branch-affect" group for branch $b_{2,1}$ and three test cases from the "branch-affect" group for branch $b_{2,2}$. All these groups are listed in Table VII. The same set of operations *BranchRelOP(b)* affects branches $b_{2,2}$, $b_{2,3}$, $b_{2,4}$ and the rest of the branches within the loop 1 for iteration 2, 3, 4, and so on. In this example, the test cases for the branch $b_{2,2}$ do not change the branch outcomes, i.e., they are always *false*. Therefore, branches within the loop 1 will always have the same outcome, so there is no need to test for infinite iterations in loop 1.

Experiment 4: JoBo [37]

JoBo is similar to jNetMap. Experiment 4 shows that our proposed method generates a finite number of test cases, while existing methods generate an infinite number of test cases.

Experiment 5: Apache Derby [42]

Apache Derby is a database written in Java. It has a higher degree of concurrency because it has more non read-shared variables. In such a program, our proposed method proves its significance because there are more potential concurrent

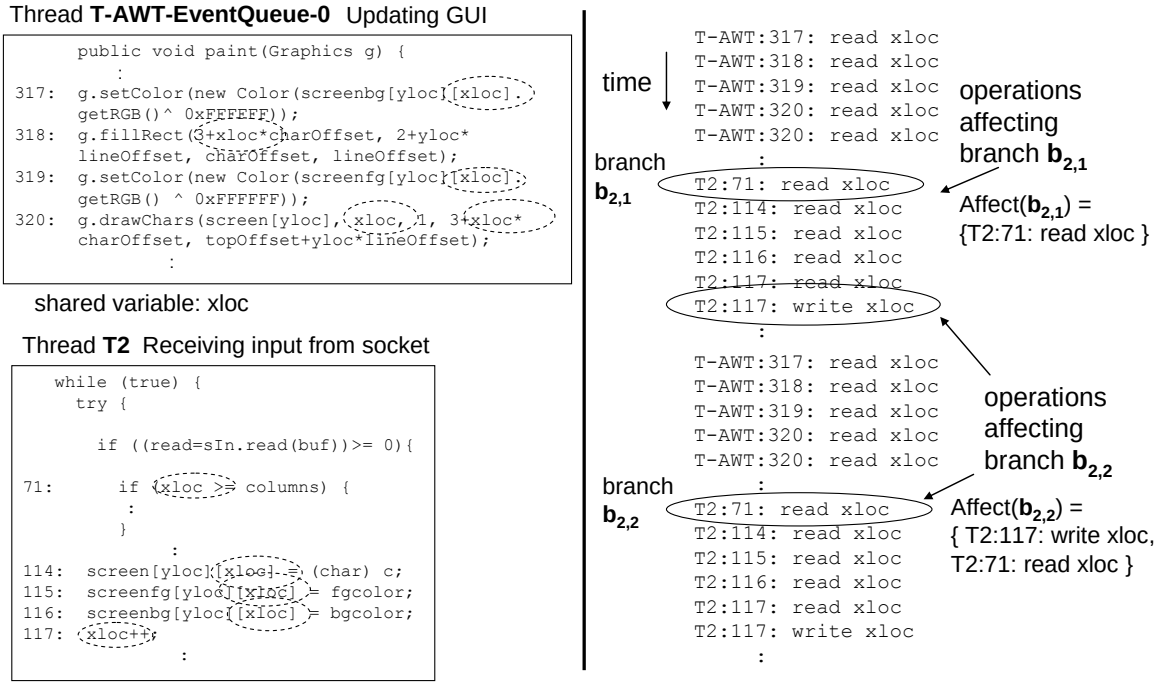


Fig. 17. The source code of the JTelnet and its execution trace.

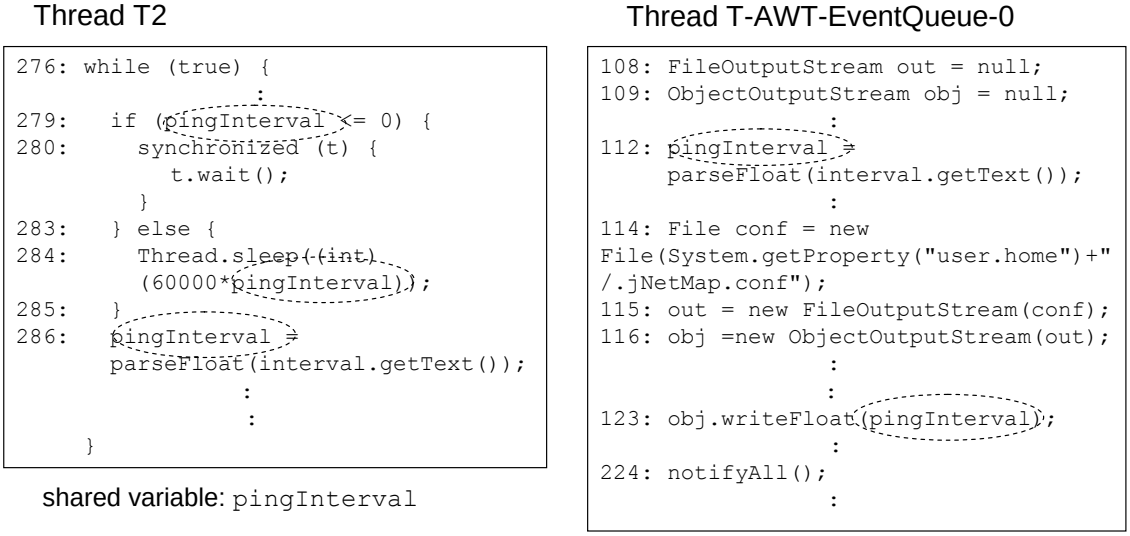


Fig. 18. The source code of the jNetMap.

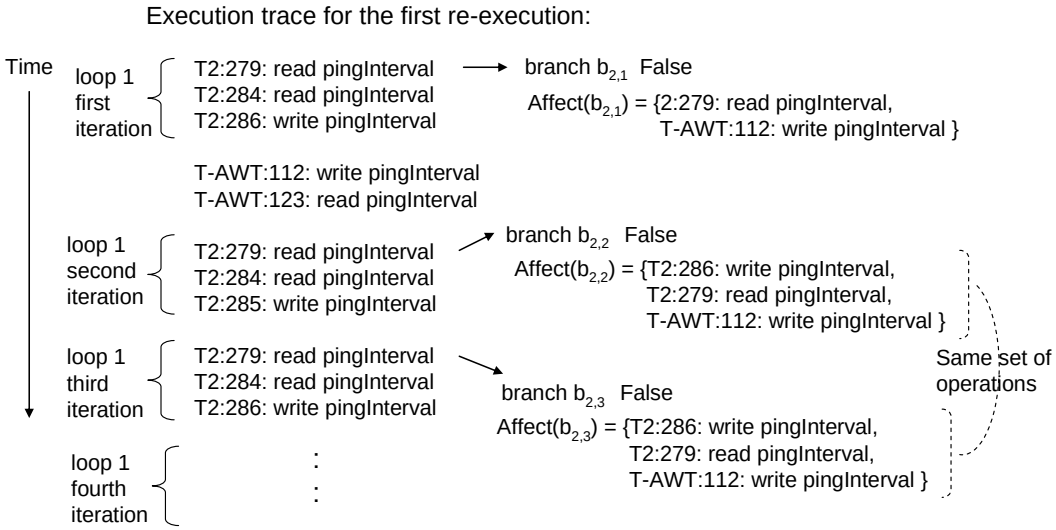


Fig. 19. Execution trace for jNetMap.

TABLE VII
"BRANCH- AFFECT" GROUPS FOR JNETMAP

"Branch- affect" groups	Order of operations from *BranchRelOP(b)*
$g1(b_{2,1})$	*T2*:279: read *pingInterval* → *T-AWT*:112: write *pingInterval*
$g2(b_{2,1})$	*T-AWT*:112: write *pingInterval*, *T2*:279: read *pingInterval*
$g1(b_{2,2})$	*T2*:286: write *pingInterval* → *T2*:279: read *pingInterval* → *T-AWT*:112: write *pingInterval*
$g2(b_{2,2})$	*T2*:286: write *pingInterval* → *T-AWT*:112: write *pingInterval* → *T2*:279: read *pingInterval*
$g3(b_{2,2})$	*T-AWT*:112: write *pingInterval* → *T2*:286: write *pingInterval* → *T2*:279: read *pingInterval*

errors that are difficult to reproduce.

VI. DISCUSSION

The usefulness of the proposed method depends on the structure of the target programs. The proposed method is useful for reproducing errors efficiently in a concurrent program which has complex lock sequences in branches. Such complex structures often make it difficult to reproduce concurrent errors because different execution paths caused by different interleavings often execute different lock sequences and accesses to shared variables. Our proposed method significantly reduces the number of test cases, first by grouping together different interleavings that do not affect consistent locking using the concept of "race-equivalence", and then by testing only one member of each group. Some concurrent programs only have read-shared variables [1], for example BlueJ [31] and Baralga [32]. The values of read-shared variables are only assigned once during initialization and they are not affected by different interleavings. Hence, they also do not have branches that are affected by different interleavings. We do not include them in our case studies because debugging such programs is relatively easy by treating them as similar to sequential programs.

Currently, our proposed method is applied to the actual target program written in Java language. Another existing work from [41] proposed prototyping for software testing and showed its benefits. Applying our method to a prototyping language could be the direction for further research.

VII. CONCLUSION

In this paper, we proposed an efficient algorithm for generating test cases for detecting concurrent program errors, particularly race conditions. The proposed method is intended as a complement for dynamic race detector tools. We extended past work, in particular [11] which concerned reachability testing, to improve efficiency for detecting race conditions by reducing the number of required test cases. The originality of our proposed method represents an improvement in efficiency in the following ways:

1. *Reduction of test cases that do not affect consistent locking for accessing shared variables*. The existence of race conditions in concurrent programs is detected by checking the consistent locking for access to shared variables among threads. In this sense, interleavings that do not change the concurrent execution path in a thread produce redundancy with respect to checking race conditions because they will have the same consistent locking. Therefore, for the detection of race conditions we can classify them into the same "race-equivalent" group and check only one from each group. Since a concurrent execution path in a thread is affected by branches, our proposed method identifies only those interleavings that affect branch outcomes, whereas the existing methods try to identify all interleavings which may affect shared variables. Our proposed method identifies only those interleavings that affect branch outcomes by utilizing data flow from the trace information to identify redundancy. For identifying data flow, we use an extension of the notation "use-define" to cover the usage and definition of shared variables in multi-thread programs. We first identify the set of operations that affect the conditional statements of branches. Based on this analysis, we can determine which interleavings affect the branches' outcomes. This significantly reduces the number of different interleavings needed for testing.
2. *Reduction of test cases by eliminating infeasible interleavings*. Our method extends the existing model of variant graphs to identify infeasible interleavings caused by lock-unlock and wait-notify operations.

We conducted some experiments with several existing Java concurrent programs and demonstrated the effectiveness of our proposed method. The experiments' results suggest that redundant interleaving can be identified and removed and that our method leads to a significant reduction in the number of test cases.

REFERENCES

[1] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*. 1997.

[2] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G, Ratsaby, and S. Ur, "Framework for testing multi-threaded Java programs," *Concurrency and Computation: Practice and Experience. John Wiley & Sons*, 2003; 15(3-5): pp. 485-499.

[3] Digital Equipment. AltaVista Search. Available: http:/altavista.digital.com/, 1996.

[4] Digital Equipment. Vesta Home Page. http:/www.research.digital.com/SRC/vesta/, 1996.

[5] E. K. Lee, C. Thekkath, and A. Petal, "Distributed virtual disks," *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VII), 1996.

[6] J.D. Choi and H. Srinivasan, "Deterministic Replay of Multithreaded Java Applications," *ACM SIGMETRICS SPDT98*, Oregon, August 1998.

[7] M.C. Baur, "Instrumenting Java bytecode to replay execution traces of multithreaded programs," *Formal Methods Group, Computer Systems Institute, Swiss Federal Institute of Technology* (ETH Zurich). 2003.

[8] R.E. Prather and J. P. Myers Jr, "The path prefix software testing strategy," *IEEE Transactions on Software Engineering*. vol. SE-13, issue: 7. July 1987.

[9] A. Bron, E. Farchi, Y. Magid, Y. Nir, S. Ur, "Applications of Synchronization Coverage," *Principles and Practice of Parallel Programming, Proceedings of the tenth ACM SIGPLAN symposium*

on Principles and practice of parallel programming. Chicago, IL, USA, 2005; pp. 206 - 212.
[10] C. Yang, A.L. Souter, L.L. Pollock, "All-du-path coverage for parallel programs," *International Symposium on Software Testing and Analysis*, 1998; pp. 153-162.
[11] G. Hwang, K. Tai, T. Huang, "Reachability Testing: An approach to testing concurrent software," *International Journal of Software Engineering and Knowledge Engineering*. 1995.
[12] P. Godefroid, "Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem," *Lecture Notes in Computer Science. Springer-Verlag*, vol. 1032, January 1996.
[13] E.M. Clarke, O. Grumberg, A.P. Doron, *Model checking*. The MIT Press. January 2000.
[14] P. Godefroid, "Model checking for programming languages using VeriSoft," *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris. January 1997.
[15] H. Nishiyama, "Detecting data races using dynamic escape analysis based on read barrier," *In Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.
[16] C. Praun and T. Gross, "Object race detection," *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, 2001; pp. 70–82.
[17] M. Christiaens, K. De. Bosschere, "TRaDe, a topological approach to on-the-fly race detection in Java programs," *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM)*, April 2001.
[18] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, S. Ur, "Testing multi-threaded Java programs," *IBM System Journal, Special Issue on Software Testing*, February 2002.
[19] M. Musuvathi,S. Qadeer, and T. Ball, "CHESS: A systematic testing tool for concurrent software," *Microsoft Research Technical Report.* MSR-TR-2007-149, 2007.
[20] B. Lindstrom, P. Pettersson, J. Offutt, "Generating trace-sets for model-based testing," *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, 2007; pp. 171-180.
[21] R. Carver and Y. Lei, "A general model for reachability testing of concurrent programs," *International Conference on Formal Engineering Methods*, November 2004; pp. 76-98.
[22] Y. Lei and R.H. Carver, "Reachability testing of concurrent programs," *IEEE Transactions on Software Engineering*, vol. 32, issue 6, June 2006; pp. 382 - 403.
[23] R. Caballero, C. Hermanns, H. Kuchen, "Algorithmic debugging of Java programs," *Electronic Notes in Theoretical Computer Science* 177. 2007; pp. 75-89.
[24] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access interleaving invariants," *In ASPLOS*, 2006.
[25] J.D. Gradecki and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Willey & Sons (Asia) Pte Ltd. 2003.
[26] PLDI Experimental Results. 2006. Available: http://www.cc.gatech.edu/~mnaik7/research/pldi06_results.html
[27] TotalView Technologies. Getting Started with Replay Engine. 2010. Available: http://www.totalviewtech.com/support/documentation/pdf/ReplayEngine1-7_GettingStarted.pdf
[28] jNetMap. June 2009. Available: http://www.rakudave.ch/?q=jnetmap
[29] Daniel Kristjansson. JTelnet. 2003. Available: http://mrl.nyu.edu/~kristja/jtelnet.html
[30] J.D. Choi, B.P. Miller, R.H.B Netzer, "Techniques for debugging parallel programs with flowback analysis," ACM Transactions on Programming Languages and Systems, 1991; 13(4) pp. 491–530.
[31] BlueJ - The interactive Java environment. 2009. Available: http://www.bluej.org/
[32] Baralga. 2010. Available: http://baralga.origo.ethz.ch/
[33] F. Sebek, "Instruction cache memory issues in real-time systems," *Technology Licentiate Thesis. Computer Architecture Lab. Department of Computer Science and Engineering*. Malardalen University. Vasteras, Sweden, ISBN 97-88834-38-7, October 2002.
[34] C.S. Yang, L. Pollock, "An algorithm for all-du-path testing coverage of shared memory parallel programs," *Asian Test Symposium*, 1997; pp. 263-268.
[35] L. Wang L and S.D. Stoller, "Runtime analysis of atomicity for multi-threaded programs," *IEEE Transactions on Software Engineering*, vol. 32, issue 2, ISSN 0098-5589, February 2006; pp. 93-110.
[36] W. Pugh and N. Ayewah, "Unit testing concurrent software," *Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering*, Atlanta, November 5-9, 2007; pp. 513-516.
[37] D. Matuschek, JoBo: web spider. Dec 2006. Available at: http://www.matuschek.net/jobo-menu/
[38] J. Huang, J. Zhou, and C. Zhang, "Scaling Predictive Analysis of Concurrent Programs by Removing Trace Redundancy," *ACM Transactions on Software Engineering and Methodology*, vol. 22, issue 1. 2011.
[39] Y. Yu, T. Rodeheffer, W. Chen, "RaceTrack: efficient detection of data race conditions via adaptive tracking," *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.
[40] C. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient Data Race Detection for Distributed Memory Parallel Programs," *SC11*, November 12-18, 2011, Seattle, Washington, USA Copyright 2011 ACM 978-1-4503-0771-0/11/11
[41] L. Yu, "Prototyping, Domain Specific Language, and Testing," *Engineering Letters, International Association of Engineers (IAENG)*, vol. 16 issue 1. 19 February 2008.
[42] Apache Derby. Available: http://db.apache.org/derby/
[43] R.H.B. Netzer and B.P. Miller, "Improving the accuracy of data race detection," *In Proceedings of the Conference on the Principles and Practice of Parallel Programming*, 1991.
[44] K. Sen and G. Agha, "Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols," *UIUC Technical Report*. Department of Computer Science, January 2006.
[45] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," *In CAV. Springer*, 2006; pp. 419-423.
[46] Apache Common Pool. 2006. Available: http://jakarta.apache.org/commons/pool/
[47] S. Lu, W. Jiang, Y. Zhou, "A study of interleaving coverage criteria," *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software*, 2007.
[48] M .Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," *Proceeding PLDI '06 Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, vol. 41 issue 6, ISBN:1-59593-320-4, June 2006.

Theodorus Eric Setiadi. He received his Engineering Degree in Electrical Engineering and a Masters Degree in Computer System Engineering from the Institute of Technology, Bandung, Indonesia, in 2000 and 2002, respectively. He is pursuing his PhD degree at the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. His research interests are debugging systems and execution trace analysis.

Akihiko Ohsuga. He received a B.S. degree in mathematics from Sophia University in 1981 and a Ph.D. Degree in Electrical Engineering from Waseda University in 1995. From 1981 to 2007, he worked with the Toshiba Corporation. Since April 2007, he has been a professor in the Graduate School of Information Systems, University of Electro-Communications. His research interests include agent technologies, formal specification & verification, and automated theorem proving. He is a member of the IEEE Computer Society (IEEE CS), the Information Processing Society of Japan (IPSJ), the Institute of Electronics, Information and Communication Engineers, and the Japan Society for Software Science and Technology. He is currently a vice chair of the IEEE CS Japan Chapter. He received the 1986 Paper Award from the IPSJ.

Mamoru Maekawa. He pursued his university education at Kyoto University (BS) and the University of Minnesota (MS and PhD). He has vast experience in both the research and development of operating systems (with Toshiba, Japan and also with an American software company), as well as in teaching computer and information science to both undergraduate and graduate levels at several universities (University of Iowa, University of Texas at Austin, University of Tokyo, University of Electro-Communications). He has published more than 30 books including "Operating Systems: Advanced Concepts" (Benjamin/Cummings/Addison Wesley) and many titles in areas covering operating systems, software design and development, multimedia and artificial intelligence in the Iwanami book series.