Efficient Test Case Generation for Detecting Race Conditions

Theodorus E. Setiadi, Akihiko Ohsuga, and Mamoru Maekawa

***Abstract*— Non-deterministic behaviors of concurrent programs sometimes produce errors that depend on interleavings, such as race conditions. Unfortunately, executing all possible interleavings is usually not feasible because of their potential huge number. In our previous work, we used the existing reachability testing method to generate test cases and succeeded in reducing the number of test cases by eliminating redundant and infeasible ones. In this paper, we propose three further improvements. The first is to reduce the memory space required for generating test cases. We propose a new method by analyzing data dependency to generate only those test cases that might affect sequences of locks and shared variables. The second improvement is to generate test cases for detecting race conditions caused by accesses through reference variables. Our method can generate test cases for detecting such race conditions by creating test cases based on the data dependency of the reference variables. The third improvement is to reduce the effort involved in checking race conditions by utilizing previous test results. The new method can identify only those parts of the execution trace in which the sequence of locks and shared variables might be affected by a new test case, thus necessitating that race conditions be rechecked only for those affected parts.**

***Index Terms*— error detection, race condition, concurrent program, execution trace.**

I. Introduction

MULTI-CORE processors are now used in various computer systems ranging from super computers to PCs, and even to small cellular phones. Concurrent programming plays a very important role in fully exploiting the capability of multi-core processors for improving their performance. One of the problems in concurrent programming is to ensure data consistency. In this paper, we focus on race conditions. Race conditions can be detected from execution traces. Some execution trace analysis techniques use lockset analysis [1] [2] [3] for dynamically detecting race conditions. They verify the consistency of locking for accesses on shared variables. Most research in this field focuses on reducing false positives [4]–[7].

The execution of a sequential program depends only on input values. However, the execution of a concurrent program depends on both input values and interleavings. Race conditions cannot always be detected during testing because their occurrences depend on interleavings. In a concurrent program, a branch can take a different execution path due not only to a different input value, but also to a different interleaving. This situation happens when the program's conditional statement depends on shared variables and the shared variables are affected by interleavings. A change of branch outcomes can affect the sequence of locks and shared variables, thus affecting the occurrence of race conditions. Hence, an execution trace might contain race conditions that depend on the branches and interleavings. As such, we must consider all possible interleavings during test case generation. Unfortunately, blindly executing all possible interleavings is not usually feasible because of their huge number. Two major issues in testing concurrent programs are efficiency and precision.

Some existing work has tried to reduce the number of execution traces. For example, J. Huang, J. Zhou and C. Zhang [8] identified that frequently a large number of events recorded in an execution trace are mapped to the same lexical statements in the source code. However, removing them from execution traces might cause false negatives when checking for race conditions. Such a situation happens when a number of events from the same lexical statement in the source code are affected by a conditional statement in a branch whose "then" and "else" statements have different sequences of locks and shared variables. Another work by C. Park, K. Sen, P. Hargrove, and C. Iancu [9], known as active testing, generates a set of tuples that represents potential concurrent errors, by performing imprecise dynamic analysis of an execution trace. The later phase re-executes the program by actively controlling the thread schedule to confirm the concurrent errors. However, the set of tuples might be incomplete if some tuples were not executed in the previous execution. This situation happens when the executions of some tuples depend on the "then" or "else" statements of a branch whose conditional statement is affected by interleavings. This incomplete set of tuples might cause some false negatives for detecting race conditions.

Some existing methods determine which interleavings are to be tested based on certain criteria. The simplest one is to execute different interleavings randomly, but this method does not guarantee that errors will be detected. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur [10] improved random methods by using a heuristic approach for reducing the search space. ConTest [11] improves heuristic test case generation by using coverage criteria as a guide.

Manuscript received October 08, 2013; revised March 15, 2014.

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).

However, this does not ensure that errors will be detected because not all possible execution paths might be tested. CHESS [12] generates all interleavings of a given scenario written by a tester based on a fair scheduling. Koushik Sen and Gul Agha [13] [14] explored different execution paths by generating new interleavings as well as new input. Their tool, known as "jCute", generates all possible interleavings based on previous executions by changing the order of thread executions, starting from the smallest indexed thread. Nevertheless, here redundancy exists for detecting race conditions, because not all of the generated interleavings will change the sequences of locks and shared variables.

Coverage criteria are useful as a guide for improving efficiency in testing. They are mainly classified based on data flow, concurrent states, and control flow. Coverage criteria for concurrent programs are mostly extensions from sequential programs. "Define-use" coverage is a coverage criterion based on data flow. The extension of "define-use" for concurrent programs was presented by [15]–[17]. Another data flow coverage criterion considers the order of data dependent operations which affect the values of shared variables [18]. Yet another one is based on concurrent states [19]. This coverage criterion measures the number of concurrent states visited during an execution of a program. Other coverage criteria are derived from existing ones by partially selecting only some program components. For example, such partial selection may pertain only to some threads [20], variables, synchronization operations [21], or operations based on temporal order relations [22]. All of the coverage criteria discussed above are mainly based on program structure, but they lack considerations about race conditions.

Path coverage is another coverage criterion based on control flow. Detecting race conditions requires path coverage for checking all possible sequences of locks and shared variables. T. E. Setiadi, A. Ohsuga, and M. Maekawa [23] used the reachability testing method [24] to generate different interleavings for exploring different execution paths in concurrent programs. They focused on efficiency and succeeded in drastically reducing the number of test cases for detecting race conditions in the following ways:

-- By grouping test cases with the same locking structure, then testing only one of them.

-- By eliminating infeasible test cases caused by synchronization mechanisms, such as a wait-notify mechanism. However, some problems still remain:

-- Redundant test cases are still generated: Test cases with the same locking structure are grouped together and only one of them is tested. Therefore, the number of test executions is decreased, but some redundant test cases are still generated even though there is no need to execute them.

-- Race conditions caused by accesses through reference variables might not be detected: Their work focused only on the detection of race conditions caused by different sequences of locks and shared variables when different interleavings change branch outcomes. Actually, race conditions can also arise in different executions with the same branch outcome. Such race conditions might come about when a different inverleaving causes a reference variable to refer to different objects. A reference variable is a variable that refers to an object in Java language, and is similar to a pointer in C language. A similar situation also occurs when a lock variable refers to different lock objects.

-- Unnecessary checks for race conditions after each test: A race detector needs to check race conditions for the whole execution trace every time a new test case is executed. For most cases, this is not always necessary as it is sufficient to check for only some parts whose consistent locking might be affected by the new test case.

In this paper, we propose the following further refinements and improvements:

-- Avoiding the generation of redundant test cases: We propose to do this by exploiting data dependency to generate only those test cases that might affect sequences of locks and shared variables. Our new proposed method requires smaller sized graphs for generating test cases compared to the existing reachability testing method. This means the required memory space is reduced.

-- Generating test cases to check consistent locking for accesses through reference variables: In addition to race conditions caused by a change in branch outcomes, our proposed method can also generate test cases for checking race conditions caused by accesses through reference variables.

-- Reducing the effort involved in checking race conditions: We propose a method to identify only the parts of the execution trace whose sequences of locks and shared variables might be affected by a new test case. Race conditions are then checked again only for those affected parts. For other (unaffected) parts, we can reuse the results from previous executions, thereby also reducing the effort involved in checking race conditions.

This paper is organized as follows. In Section II, we briefly summarize the terms and notations used in the paper. Section III explains the proposed method for an efficient test case generation for detecting race conditions. Section IV shows the efficiency of our proposed method through some experiments, while Section V presents some discussions and future work. Finally, Section VI concludes the paper.

II. Terms and Notations

We briefly summarize the essential terms and notations which were originally introduced in T. E. Setiadi, A. Ohsuga, and M. Maekawa [23].

A. *"Access-Manner"*

We divide an execution path of a single thread into several parts called "access-manners". We assume a concurrency control using a lock mechanism. In order to define an "access-manner", we use the notation *L(Ti)* as the number of active locks acquired by a thread *Ti* at a particular time. *L(Ti)* is 0 at the beginning of the execution of the thread *Ti*. During an execution of a program, *L(Ti)* is incremented and decremented by the following rules:

-- Incremented by 1 when the thread *Ti* successfully acquires a lock (i.e. has completed a lock instruction).

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

An individual "access-manner" is a sequence of lock-unlock and read-write operations on shared variables within an execution path of a thread. It starts and ends with the following conditions:

-- Start: a lock operation which causes *L(Ti)* to become 1.

-- End: an unlock operation which causes *L(Ti)* to become 0, or when an execution trace terminates.

An individual "access-manner" must end before another individual "access-manner" starts; thus they cannot overlap. We classify "access-manners" based on their sequences of lock-unlock and read-write operations on shared variables as follows:

-- A usual "access-manner": starts by acquiring a lock, accessing shared variables, then releasing the corresponding lock.

-- An unusual "access-manner": starts by accessing shared variables without previously acquiring any locks, or when executing only an unlock operation without previously acquiring a lock. This might happen because programmers forget to acquire locks. Such an unusual "access-manner" might potentially cause race conditions should another thread be accessing the same shared variable. Throughout this paper, "access-manner" should be understood to mean a usual "access-manner".

B. "Use-Define"

A "use-define" is a relation consisting of a usage "use" of a variable and the definition "define" of the variable.

-- A "use" means a read operation on a variable.

-- A "define" means a write operation of some value to a variable. A "use-define" is a triplet:

$$ud(var,\ use\_location,\ define\_location) \qquad (1)$$

The "use-define" was initially defined for sequential programs. We call the "use-define" for sequential programs the conventional "use-define". Yang, A.L. Souter, and L.L. Pollock [10] [34] extend the definition of "use-define" to the usage and definition of shared variables in concurrent programs. Below are the differences:

-- Sequential program: the "use" and "define" operations are located in the same thread. There must be no other write operations to the variable in between the "use" and "define" operations.

-- Concurrent program: the "define" operation might be located in a different thread to the "use" operation. The interleaving in a particular execution decides which thread actually defines the value.

A set of "use-defines" is obtained from an execution trace. We use the set to find operations which affect conditional statements in branches or reference variables in "access-manners". From an existing "use-define", we also define another potential "use-define" for the same "use" of the variable when there could be another interleaving which satisfies the following two conditions:

-- There is another "define" operation which occurs before the "use" operation. We assume the "use" operation can be executed after the "define" operation, i.e. not blocked by a thread creation or a wait-notify message.

-- There is no other "define" operation to the variable between the "define" operation in condition 1 and the "use" operation.

A potential "use-define" is denoted by:

$$ud'(var,\ use\_location,\ define\_location) \qquad (2)$$

Figure 2 is an example of one of the possible execution traces for the source code in Figure 1. Its "use-defines" and potential "use-defines" are as follows:

-- "Use-defines": $ud(x, 3, 1)$, $ud(y, 3, 2)$, $ud(x, 25, 20)$, $ud(n, 4, 3)$, $ud(ref2, 27, 23)$

-- Potential "use-defines": $ud'(x, 3, 20)$, $ud'(x, 25, 1)$, $ud'(ref2, 27, 30)$

Let *setUD(V)* be the set of "use-defines" in an "execution-variant" *V*. An "execution-variant" *V* satisfies a "use-define" *ud*(*var*, *use_location*, *define_location*) if the "use-define" is included in the *setUD(V)*. In other words, it satisfies the following condition:

$$ud(var,\ use\_location,\ define\_location) \subseteq setUD(V) \qquad (3)$$

Let *define_set*(*var*, *use_location*) be the set of possible "define" operations for the variable *var* at the location *use_location*. Below are some examples of *define sets* in Figure 2:

-- $define\_set(x, 3) = \{\ 1{:}\ x = -3,\ 20{:}\ x = 10\ \}$

-- $define\_set(y, 3) = \{\ 2{:}\ y = 2\ \}$

-- $define\_set(n,4) = \{\ 3{:}\ n = x + y\ \}$

-- $define\_set(x, 22) = \{\ 1{:}\ x = -3,\ 20{:}\ x = 10\ \}$

If a *define set* contains only one "define" operation from the same thread, then we can guarantee that its values will not be affected by different interleavings.

C. Variant Graph and "Execution-Variant"

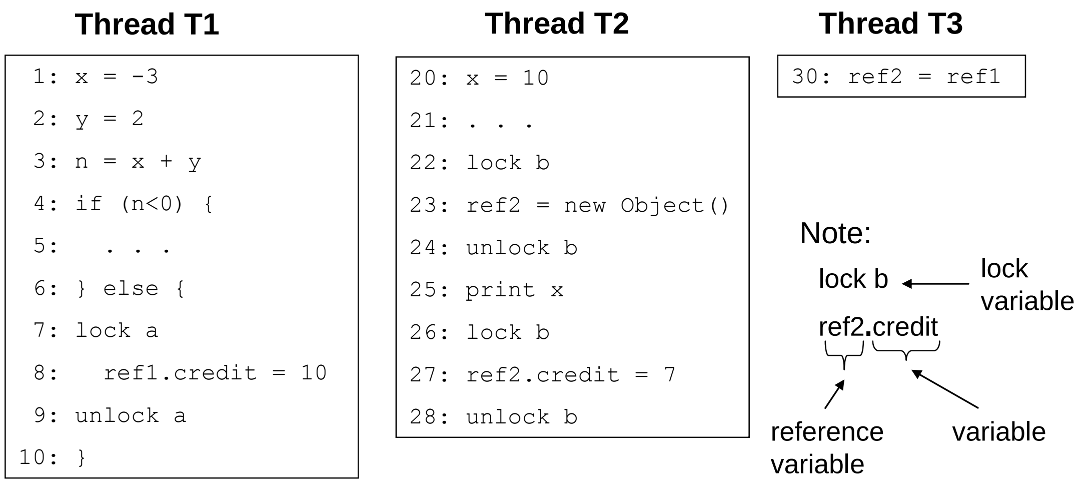
The reachability testing method [24] performs an efficient exploration of different sequences of read-write operations which affect values of shared variables. Using the idea behind the partial order reduction, it groups and ignores different interleavings that do not affect any values of shared variables. Test cases are generated systematically using a variant graph. A variant graph derives different sequences of read-write operations from the previous execution trace. A different sequence of read-write operations which affects the values of shared variables is called an "execution-variant". "Execution-variants" are used as test cases in the reachability testing method. G. H. Hwang, K. C. Tai, and T. L. Huang introduced an algorithm to create a variant graph from an execution trace of a concurrent program (Hwang et al., 1995). Figure 3 is an example of a variant graph for the execution trace in Figure 2.

III. Proposed Method

A. System Overview

The proposed system is a refinement of the existing deterministic testing method with tracing and dynamic race detection. Figure 4 shows the overview of the proposed method. The whole procedure for testing is shown as follows:

1) Execute a concurrent program by taking a trace.
2) Detect branches, concurrent-pairs of "access-manners", and a set of "use-defines" from the execution trace.
3) Create concurrent dependency graphs from branches and concurrent-pairs of "access-manners". A concurrent dependency graph represents data flow relations among operations that might affect race conditions.

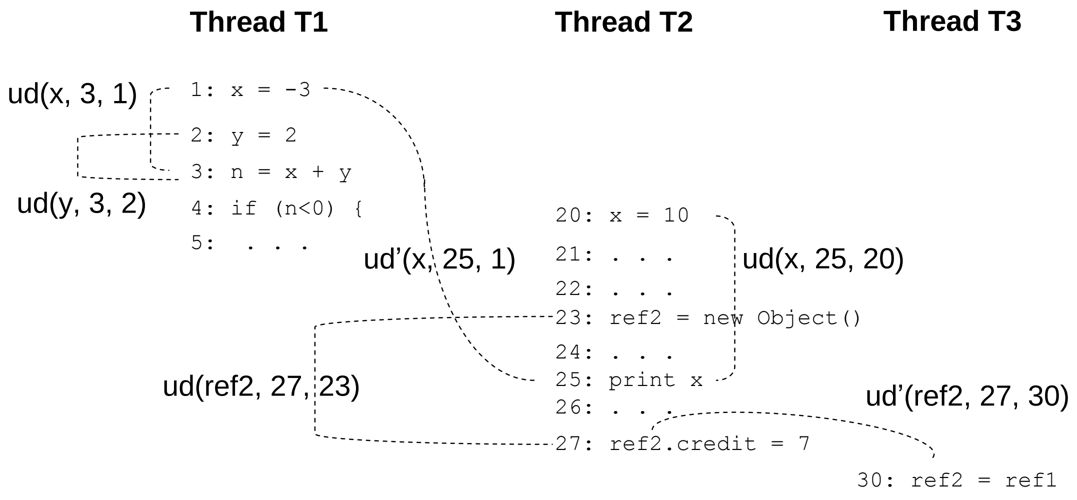


```
Thread T1
 1: x = -3
 2: y = 2
 3: n = x + y
 4: if (n<0) {
 5:   . . .
 6: } else {
 7: lock a
 8:   ref1.credit = 10
 9: unlock a
10: }
```

```
Thread T2
20: x = 10
21: . . .
22: lock b
23: ref2 = new Object()
24: unlock b
25: print x
26: lock b
27: ref2.credit = 7
28: unlock b
```

```
Thread T3
30: ref2 = ref1
```

Fig. 1. Example of a concurrent program.



```
1: x = -3
2: y = 2
3: n = x + y
4: if (n<0) {
5:   . . .
```

```
20: x = 10
21: . . .
22: . . .
23: ref2 = new Object()
24: . . .
25: print x
26: . . .
27: ref2.credit = 7
```

```
30: ref2 = ref1
```

Fig. 2. Example of an execution trace and some of its "use-defines".

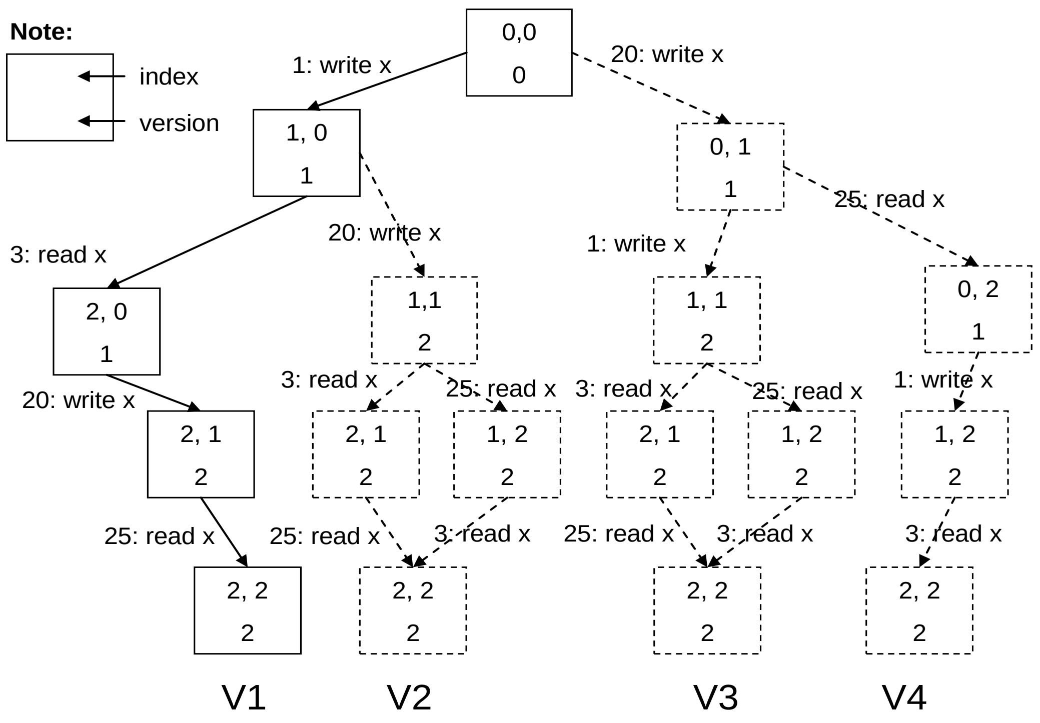


Fig. 3. Example of a variant graph.

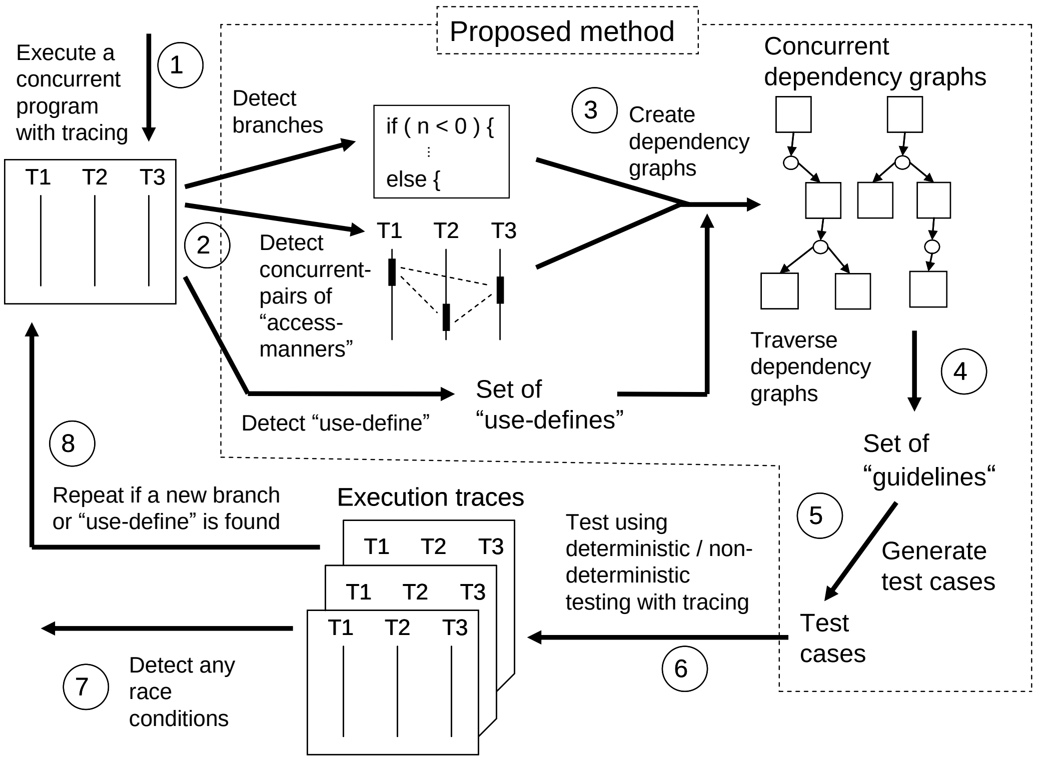

Proposed method
Execute a concurrent program with tracing
1
T1 T2 T3
Detect branches
if (n < 0) {
else {
3
Create dependency graphs
Concurrent dependency graphs
2
Detect concurrent-pairs of "access-manners"
T1 T2 T3
Detect "use-define"
Set of "use-defines"
Traverse dependency graphs
4
Set of "guidelines"
5
Generate test cases
Test cases
8
Repeat if a new branch or "use-define" is found
Execution traces
Test using deterministic / non-deterministic testing with tracing
6
7
Detect any race conditions

Fig. 4. Overview of the proposed method.

4) Determine a set of "guidelines" for generating test cases. A "guideline" is a set of "use-defines" obtained by traversing the concurrent dependency graphs from the previous step.
5) Generate test cases based on the set of "guidelines" from step 4. The idea is to generate only those test cases necessary to avoid redundancy and that do not affect race conditions.
6) Execute the test cases using an existing deterministic/non-deterministic testing method by taking a trace.
7) Detect any race conditions using an existing race detector and report any race conditions to programmers.
8) If a new branch or a new "use-define" is found in the execution trace in step 6, repeat step 3 to step 8 for the new branch or the new "use-define".
9) The test is completed if neither a new branch nor a new "use-define" is found in step 6.

Note that this new method introduces "concurrent dependency graphs", instead of variant graphs. Variant graphs are the major instruments for representing and analyzing the execution development of a concurrent program in the reachability testing method.

B. Avoiding Redundancy in Test Case Generation

This section explains how to avoid generating redundant test cases by using a concurrent dependency graph.

Concurrent Dependency Graph

We newly propose a concurrent dependency graph for identifying data dependencies of shared variables or reference variables. A concurrent dependency graph is a directed graph representing "use-define" relations in an execution of a concurrent program. A conventional dependency graph depends only on data flow, but a concurrent dependency graph depends on data flow and interleavings. A concurrent dependency graph contains all possible data dependencies for different interleavings. Which data dependency actually occurs in a particular execution would depend on the interleaving during the execution. Figure 5 shows an example of a concurrent dependency graph.

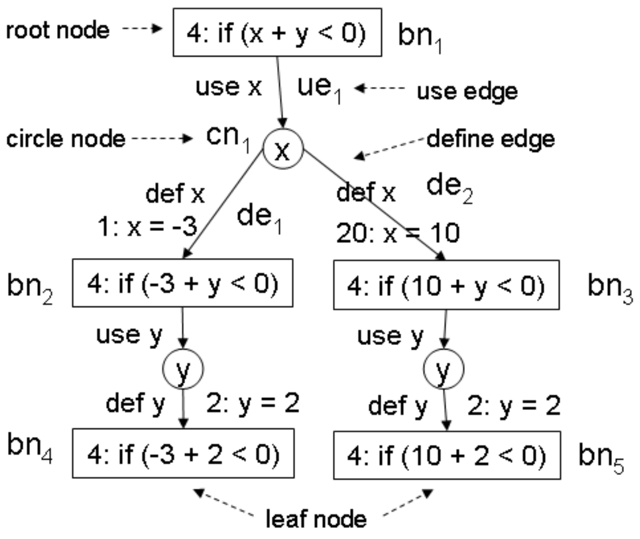


Fig. 5. Components of a concurrent dependency graph.

Let us take an example of the shared variable *x* in the root node. There are two write operations that can define its value depending on the interleavings. One is the write operation in line 1 while the other one is in line 20. The components of a concurrent dependency graph are as follows:

■ Node:
- ● Box node (*bn*):
  - ✧ Root node: represents one of the following:
    - - - A conditional statement in a branch (see the example in Figure 5), or

- - An "access-manner" (see the example in Figure 12).

A root node does not have an incoming edge.

✧ Non-root node: derived from a root node or another non-root box node. Algorithm 1 explains how to derive non-root nodes. A non-root node has one incoming and one outgoing edge.

✧ Leaf node: a box node whose statement does not contain any variables. When a variable is used without being defined, then there will be no corresponding leaf node. A leaf node does not have an outgoing edge. In Figure 5, nodes bn_4 and bn_5 are leaf nodes.

The maximum number of outgoing edges from a box node is 1.

● Circle node (*cn*): represents a selection of "define" operations for a variable.

■ Edge:

● "Use" edge (*ue*): represents a read operation to a variable. This edge goes out from a box node and comes into a circle node. It is labeled by the program statement that reads to the variable.

● "Define" edge (*de*): represents a write operation to a variable. This edge goes out from a circle node and comes into a box node. It is labeled by the program statement that writes to the variable.

Table I lists the definitions in a concurrent dependency graph. A concurrent dependency graph is created by deriving child nodes starting from their root node. Algorithm 1 explains how to derive child nodes from a box node, while Figure 6 is an illustration of Algorithm 1.

Algorithm 1. Deriving child nodes from a box node.

```
Input : - A box node $bn_{input}$ as a parent node.
        - A set of "use-defines" and potential "use-defines".
Output : - The input parent node is connected to a newly-created circle node cn as a child node.
         - The circle node cn is connected to newly-created box node(s) as its child node(s).
Step 1. Create a circle node cn for the input box $bn_{input}$.
    1.1 Choose a variable var from the statement inside the $bn_{input}$.
    1.2 Create a new circle node cn and label it as var.
    1.3 Create an outgoing "use" edge ue from the $bn_{input}$ to the circle node cn created in step 1.2.
    1.4 Label the "use" edge ue with the variable chosen in step 1.1.
Step 2. Create child nodes for the circle node cn.
    2.1 Find "define" operations for variable(cn) from the set of "use-defines".
    2.2 For every "define" operation in step 2.1, create one "define" edge de.
        2.2.1 For each "define" edge de in step 2.2, create a box node bn.
            2.2.1.1 Make the de the incoming edge for the bn.
            2.2.1.2 The box node bn contains the statement from the $bn_{input}$ with the variable var substituted by the define statement in step 2.2.
```

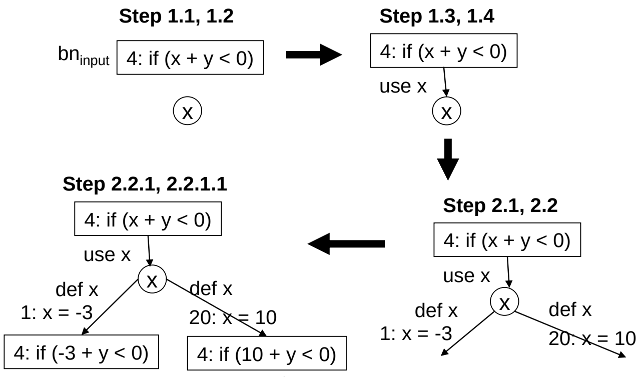


Fig. 6. Step by step illustration for Algorithm 1.

Algorithm 2 explains how to construct a concurrent dependency graph. It derives a box node using Algorithm 1 until all the derived child nodes reach leaf nodes.

Algorithm 2. Constructing a concurrent dependency graph.

```
Input: - A set of "use-defines" and potential "use-defines" from an execution trace.
       - A root node.
Output: A concurrent dependency graph dg.
Step 1. Initialization: include the root node in the concurrent dependency graph dg.
Step 2. For every box node bn in dg that does not have an outgoing edge.
    2.1 Create child nodes bn using Algorithm 1.
Step 3. Repeat step 2 until no more new edges or new boxes are created.
```

Figure 7(a) shows a concurrent dependency graph constructed using Algorithm 2 for the branch in Figure 2.

Only variables with a *define set* of more than one member within a concurrent dependency graph can create different "execution-variants". Therefore, any variables with only one member in their *define set* are redundant with respect to exploring different "execution-variants". Algorithm 3 describes how to optimize a concurrent dependency graph by removing such a redundancy. Figure 7(b) shows an example of an optimized dependency graph.

Algorithm 3. Optimizing a concurrent dependency graph.

```
Input: A concurrent dependency graph dg.
Output: An optimized concurrent dependency graph dg.
Step 1. For each circle node cn in the concurrent dependency graph dg.
    1.1 If cn has only one outgoing edge.
    Then
        1.1.1 Remove the parent node of cn and all edges connected to cn.
        1.1.2 Make the incoming edge of parent(cn) the incoming edge of child_node(cn).
Note: step 1.1.2 is not applicable if the parent(cn) is a root node, because a root node does not have an incoming edge.
```

TABLE I
DEFINITIONS IN A CONCURRENT DEPENDENCY GRAPH

Definitions	Examples (refer to Figure 5)
variable(*ue*) : the variable used by a "use" edge *ue*.	*variable*(ue_1) = *x*
variable(*de*) : the variable defined by a "define" edge *de*.	*variable*(de_1) = *x*

variable(*bn* | *cn*): the set of variables in the statement of node *bn* or *cn*.	*variable*(bn_1) = { *x*, *y* }
	variable(cn_1) = {*x*}

def_edge(*cn*): the set of define edges for a circle node *cn*.	*def_edge*(cn_1) = {de_1, de_2}
parent(*cn*): the parent node of a circle node *cn*. *parent*(*cn*) = { *bn* | where a use-edge *ue* exists in which *ue* is the outgoing edge of *bn*, *ue* is the incoming edge of *cn*, variable(*bn*) ∩ variable(*cn*) ≠ Ø }	*parent*(cn_1) = bn_1
child(*bn*): the child node of *bn*. *child*(*bn*) = { *cn* | where a "use" edge *ue* exists in which *ue* is the outgoing edge of *bn*, *ue* is the incoming edge of *cn*, *variable*(*bn*) ∩ *variable*(*cn*) ≠ Ø } Note: The child node of a box node is a circle node that represents the "use" of a variable within the statement of the box node. A box node can only have one circle node as its child node.	*child*(bn_1) = cn_1
child(*cn*): the set of child nodes of *cn*. child(*cn*) = { *bn* | where for every *bn*, a define edge *de* exists in which *de* is an outgoing edge of *cn*, *de* is an incoming edge of *bn* } Note: A circle node *cn* does not have any child nodes if the variable for *cn* is used without being defined.	*child*(cn_1) = { bn_2, bn_3}

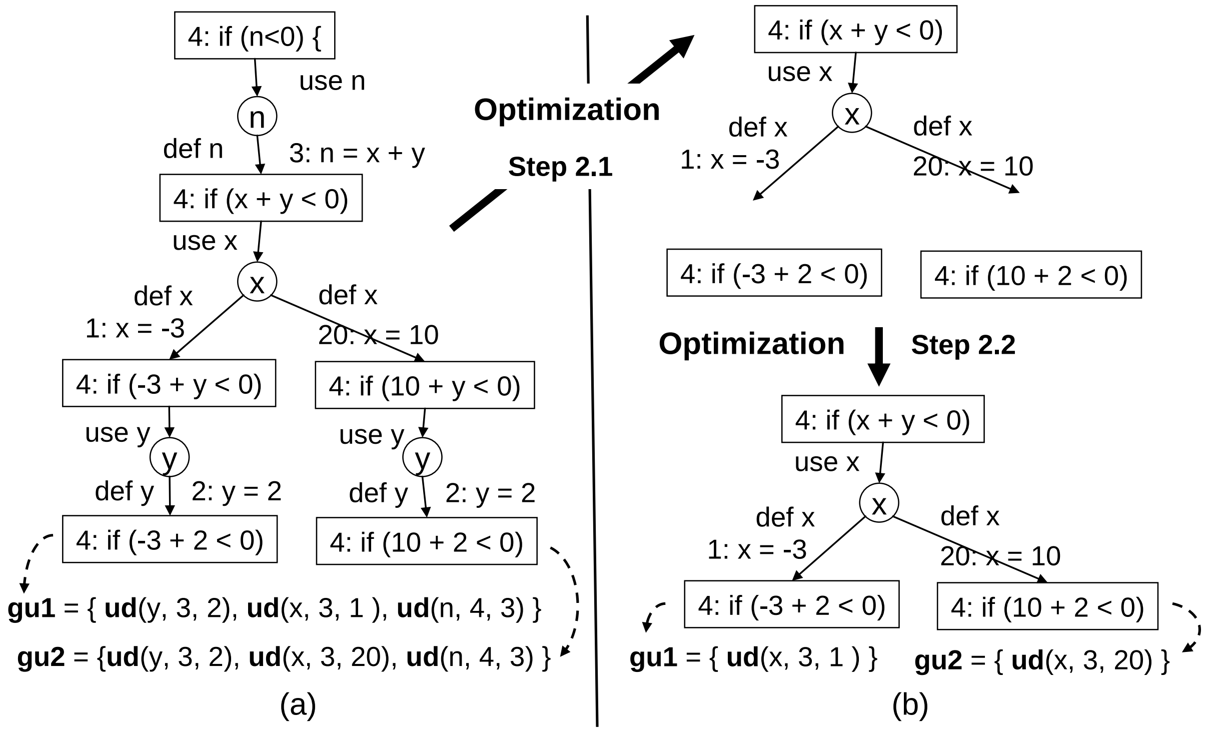


Fig. 7. Example of a concurrent dependency graph (a) and its optimized version (b).

The optimized graph is more efficient because it is smaller and thus requires fewer steps to traverse. The next subsection explains how to traverse a dependency graph.

Traversing a Concurrent Dependency Graph

A race condition can occur because different interleavings affecting branch outcomes can lead to different sequences of locks and shared variables. This subsection explains how to generate different interleavings in order to explore different branch outcomes. We use the term "guidelines" as a set of "use-defines" for generating a test case. The "guidelines" determine the data dependency for creating a test case. An "execution-variant" *V* satisfies a "guideline" if all members of the "guideline" are included in the set of "use-defines" of the "execution-variant" *V*. In other words, the following condition must be satisfied:

$$\text{all members of "guideline"} \subseteq setUD(V) \qquad (4)$$

Algorithm 4 explains how to traverse the paths in a concurrent dependency graph to obtain a set of "guidelines". Table II is an example of a set of "guidelines" obtained by applying Algorithm 4 to the concurrent dependency graph in Figure 7(a).

Algorithm 4. Traversing a concurrent dependency graph.

```
Input: A concurrent dependency graph dg.
Output: A set of "guidelines" for generating test cases.
Step 1. Initialization.
    Let the output set of "guidelines" = { Ø }
Step 2. Start from the root node of the input concurrent dependency graph dg, do a "Depth First Search " (DFS).
        2.1 When the DFS visits a leaf node, extract the set of "use-defines" from the root node to the leaf node and add them as a "guideline" to the set of "guidelines" as the output.
        2.2 Repeat step 2.1 until all leaf nodes in the concurrent dependency graph dg have been visited.
```

One test case will be created for each guideline, so there will be two test cases based on Table II. The "use-define" *ud*(*y*, 3, 2) and *ud*(*n*, 4, 3) are the same for both guidelines. They are redundant because the concurrent dependency graph in Figure 7(a) is not optimal. In order to distinguish between these two test cases, only the "use-defines" on variable *x* matter. Table III is an example of a set of "guidelines" obtained by applying Algorithm 4 to the optimized concurrent dependency graph in Figure 7(b). It shows that only the "use-defines" on variable *x* are necessary to distinguish between those two guidelines.

TABLE II
A SET OF "GUIDELINES" FROM THE CONCURRENT DEPENDENCY GRAPH IN FIGURE 7(A)

NO.	"Guideline"
1	*gu1* = { *ud*(*y*, 3, 2), *ud*(*x*, 3, 1), *ud*(*n*, 4, 3) }
2	*gu2* = { *ud*(*y*, 3, 2), *ud*(*x*, 3, 20), *ud*(*n*, 4, 3) }

TABLE III
A SET OF "GUIDELINES" FROM THE CONCURRENT DEPENDENCY GRAPH IN FIGURE 7(B)

NO.	"Guideline"
1	*gu1* = { *ud*(*x*, 3, 1) }
2	*gu2* = { *ud*(*x*, 3, 20) }

Generating Test Cases from a Concurrent Dependency Graph

This subsection explains an efficient test case generation using a set of "guidelines" from a concurrent dependency graph. We recall some definitions from the work by T. E. Setiadi, A. Ohsuga, and M. Maekawa [23] in the subsection on a Model for Concurrent Program Execution Traces about the sequence of operations in an execution of a concurrent program. These are as follows:

-- *S* is a sequence of read-write operations from an execution trace.

-- *S(j)* is a sequence of read-write operations in thread *Tj*.

-- *S(j, i)* is the *i*-th operation in the sequence of operations in thread *Tj*.

The task for generating test cases can be stated as follows: Given a concurrent dependency graph *dg* derived from an existing sequence of read-write operations *S1* and the following set of "guidelines" obtained from the concurrent dependency graph *dg*:

-- *gu1* = { *ud*(*var*, *use*, *def1*) }

-- *gu2* = { *ud*(*var*, *use*, *def2*) }

Supposing that the existing sequence of read-write operations *S1* satisfies the "guideline" *gu1*, create another sequence of read-write operations *S2* that satisfies the "guideline" *gu2*. Let:

-- *S(a,j)* = the "use" operation in the guideline *gu2*.

-- *S(a,j-1)* = one operation in the thread *Ta* before the "use" operation *S(a,j)*.

-- *S(b k)* = the "def2" operation in the guideline *gu2*.

-- *S(b,k-1)* = one operation in the thread *Tb* before the "def2" operation *S(b,k)*.

The solution for the *S2* depends on whether the "use" operation is located in the same thread as "def2" operation or not:

-- Case 1: the "use" operation is in the same thread as the "def2" operation, i.e. they are located in the same thread *Tb*, *S(b,j)* = "use" operation and *S(b,k)* = "def2" operation (refer to Algorithm 5).

-- Case 2: The "use" operation is in a different thread to the "def2" operation (refer to Algorithm 6).

Figure 8 illustrates the examples of these two cases.

Algorithm 5. Generating test cases if the "define" operation is in the same thread as the "use" operation.

```
Step 1. Select the next operation non-deterministically.
Step 2. If the operation selected in step 1 is the "def2" operation S(b, k),
    Then
        2.1 The next operations are from thread Tb until the "use" operation S(b, j).
        2.2 Select the next operations non-deterministically until the concurrent program terminates.
        2.3 Terminate this algorithm.
    Else
        2.1 Repeat from step 1.
```

Algorithm 6. Generating test cases if the "define" operation is in a different thread to the use" operation.

```
Step 1. Initialization:
        - All threads are not blocked.
```

```
Step 2. Select the next operation non-deterministically from any non-blocked threads.
Step 3. Check whether the operation selected in step 2 is one operation before the "use" operation or before the "def2" operation.
    3.1 If the operation selected in step 2 is S(b, k-1)
      Then
        3.1.1 Thread Tb is blocked.
    3.2 If the operation selected in step 2 is S(a, j-1)
      Then
        3.2.1 Thread Ta is blocked.
Step 4. If thread Ta and thread Tb are blocked
  Then
    4.1 Execute "def2" and "use" consecutively as the next operations.
    4.2 Select the next operations non-deterministically until the concurrent program terminates.
    4.3 Terminate this algorithm.
  Else
    4.1 Repeat from step 2.
```

An example of case 2:

-- From Figure 2: *S1* is T1:1:*x* = -3, T1:2:*y* = 2, T1:3:*n* = *x*+*y*, T1:4:if(*n*<0), T1:5:..., T2:20:*x* = 10, T2:21:..., T2:22:..., T2:23:*ref2* = new Object(),T2:24:..., T2:25:print *x*, T2:26:..., T2:27:*ref2.credit* = 7, T3:30:*ref2*=*ref1*

-- Figure 7(b): Let *dg* be the concurrent dependency graph derived from the existing sequence *S1*.

-- From Table III: the set of "guidelines" = { *gu1* = { *ud*(x, 3, 1) }, *gu2* = { *ud*(x, 3, 20) } } is derived from the concurrent dependency graph *dg* in Figure 7(b).

This example falls into case 2 because the "use" and "def2" in *gu2* are in different threads. Figure 9 illustrates the test case generation. The sequence for *S2* is T1:1:*x* = -3, T1:2:*y* = 2, T2:20:*x* = 10, T1:3:*n* = *x*+*y*, T1:4:if (*n*<0), T1:5:..., T2:21:..., T2:22:..., T2:23:*ref2* = new Object(),T2:24:..., T2:25:print *x*, T2:26:..., T2:27:*ref2.credit* = 7, T3:30:*ref2* = *ref1*.

Comparison with the Existing Reachability Testing Method

This subsection explains an example for test case generation using the existing reachability testing method. Figure 3 is an example of a variant graph for the execution trace in Figure 2. In this example, we exclude the shared variable *ref2* and consider only the shared variables *x* and *y* to simplify the explanation. There are four "execution-variants"; they are *V1*, *V2*, *V3*, and *V4* as shown in Figure 3. Dotted boxes in a variant graph represent some read or write operations accessing different values of shared variables as the result of different interleavings.

The variant graph in Figure 3 generates four test cases, but some of them are redundant. From the set of "guidelines" in Table II or Table III, our proposed method identifies that only two test cases are required. Table IV shows different values of variables when executing different "execution-variants". The "execution-variants" *V1* and *V3* have the same truth value for the branch in line 4. It is sufficient to test only one of them with respect to exploring different execution paths caused by the branch. They differ in the values of the variable *x* in line 25, but the truth value of the branch in line 4 is the same. A similar situation happens for the "execution-variants" *V2* and *V4*. Suppose that the "execution-variant" *V1* is executed when the program is first tested. The "execution-variant" *V2* can be created from *V1* by replacing the "use-define" *ud*(*x*, 3, 1) with *ud*(*x*, 3, 20).

TABLE IV
DIFFERENT VALUES OF VARIABLES AMONG DIFFERENT "EXECUTION-VARIANTS"

"EXECUTION-VARIANT"	3: read *x*	3: read *y*	3: write *n*	4: if (*n*<0)	25: read *x*
V1	-3	2	-1	*True*	10
V2	10	2	12	*False*	10
V3	-3	2	-1	*True*	-3
V4	-3	2	12	*False*	10

Figure 9 shows how to generate only the required test cases based on the "guideline" from the proposed concurrent dependency graph.

C. Generating Test Cases to Check Consistent Locking for Access through Reference Variables

The difficulty in detecting race conditions is not only because a different interleaving can change branch outcomes, but also because it can change a reference variable to refer to a different object. A similar situation also occurs when a lock variable refers to a different lock object. In this subsection, we show that our proposed concurrent dependency graph can also generate test cases in such a situation. An example of the situation is illustrated below:

-- In Figure 10, the truth value of the branch depends on the order of executions of the "access-manner" *M1* and *M4* as seen in Figure 10(a) and Figure 10(b). In the event that the branch takes a different execution path, the error might not be detected.

-- The reference variables *ref1* and *ref2* can refer to the same or different objects depending on the order of executions of the "access-manners" *M5* and *M6*, as shown in Figure 10(b) and Figure 10(c). A race condition arises in execution 3 in Figure 10(c) in the event that the "access-manner" *M3* and "access-manner" *M5* are not protected by the same lock. A race condition cannot bedetected in execution traces 1 or 2, but can be detected in execution trace 3.

Concurrent-pairs of "Access-Manners"

We use the term 'concurrent-pair' of "access-manners" for checking race conditions in a concurrent execution. Two "access-manners" *M1* and *M2* are a concurrent-pair, denoted by *pair(M1, M2),* if there exists a different interleaving that can change the order of occurrence between one of the operations from *M1* and one of the operations from *M2*. Let's assume an "access-manner" *M1* in a thread *T1*, and an "access-manner" *M2* in a thread *T2*. The "access-manners" *M1* and *M2* are a concurrent-pair of "access-manners" if the following three conditions hold:

-- Different threads: The threads *T1* and *T2* are different.

-- Not blocked by a thread creation: The thread *T1* is not created by the thread *T2* after the "access-manner" *M2* ends, or the thread *T2* is not created by the thread *T1* after the "access-manner" *M1* ends.

-- Not blocked by a synchronization message: The thread *T1* does not wait for a message from the thread *T2* before the "access-manner" *M1* starts, or the thread *T2* does not wait for a message from the thread *T1* before the "access-manner" *M2*

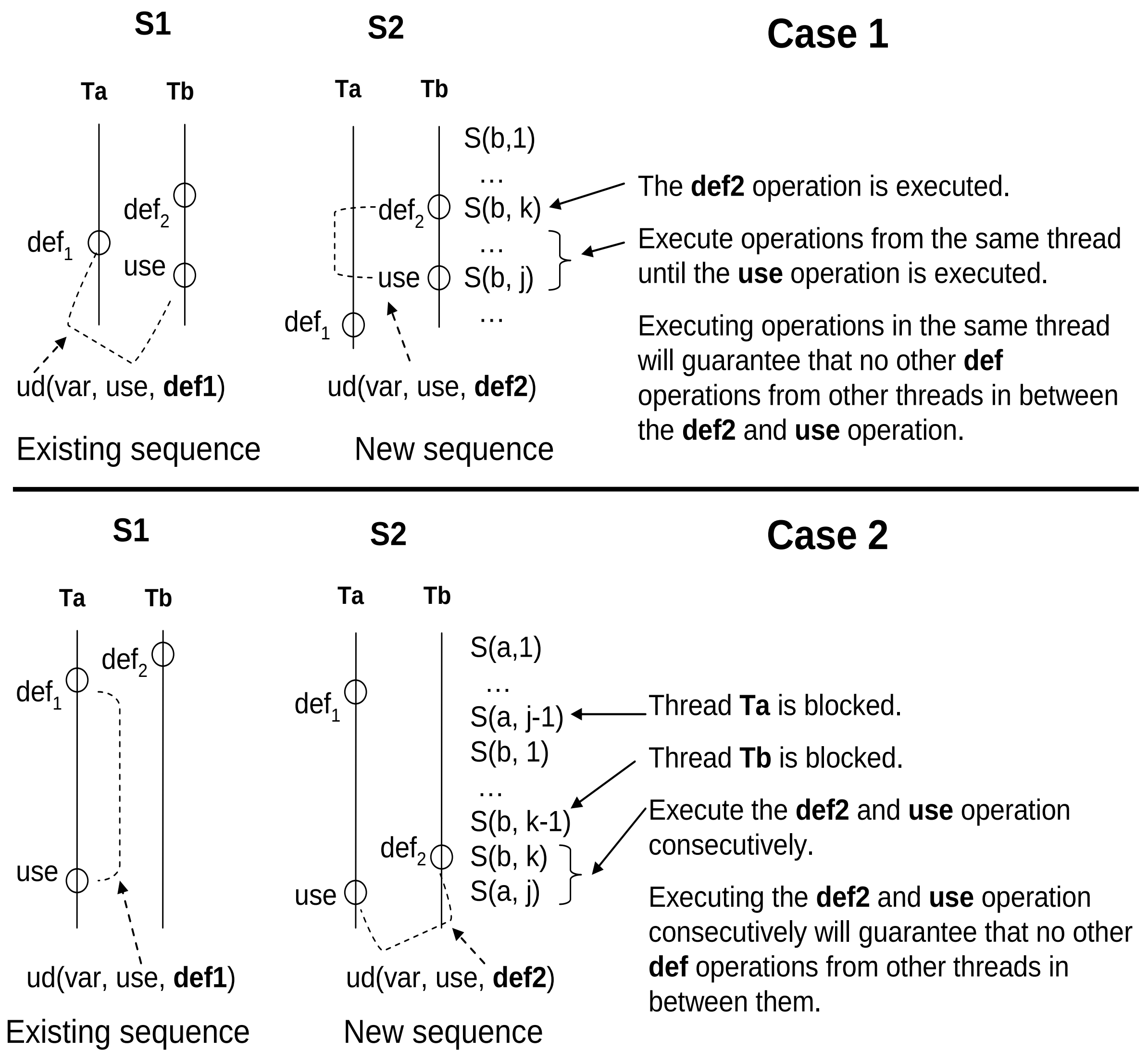

Case 1
S1
Ta
Tb
def1
def2
use
ud(var, use, def1)
Existing sequence
S2
Ta
Tb
S(b,1)
…
def2
S(b, k)
…
use
S(b, j)
def1
…
ud(var, use, def2)
New sequence
The def2 operation is executed.
Execute operations from the same thread until the use operation is executed.
Executing operations in the same thread will guarantee that no other def operations from other threads in between the def2 and use operation.
Case 2
S1
Ta
Tb
def2
def1
use
ud(var, use, def1)
Existing sequence
S2
Ta
Tb
S(a,1)
…
def1
S(a, j-1)
S(b, 1)
…
S(b, k-1)
def2
S(b, k)
use
S(a, j)
ud(var, use, def2)
New sequence
Thread Ta is blocked.
Thread Tb is blocked.
Execute the def2 and use operation consecutively.
Executing the def2 and use operation consecutively will guarantee that no other def operations from other threads in between them.

Fig. 8. Example of test case generation for different cases.

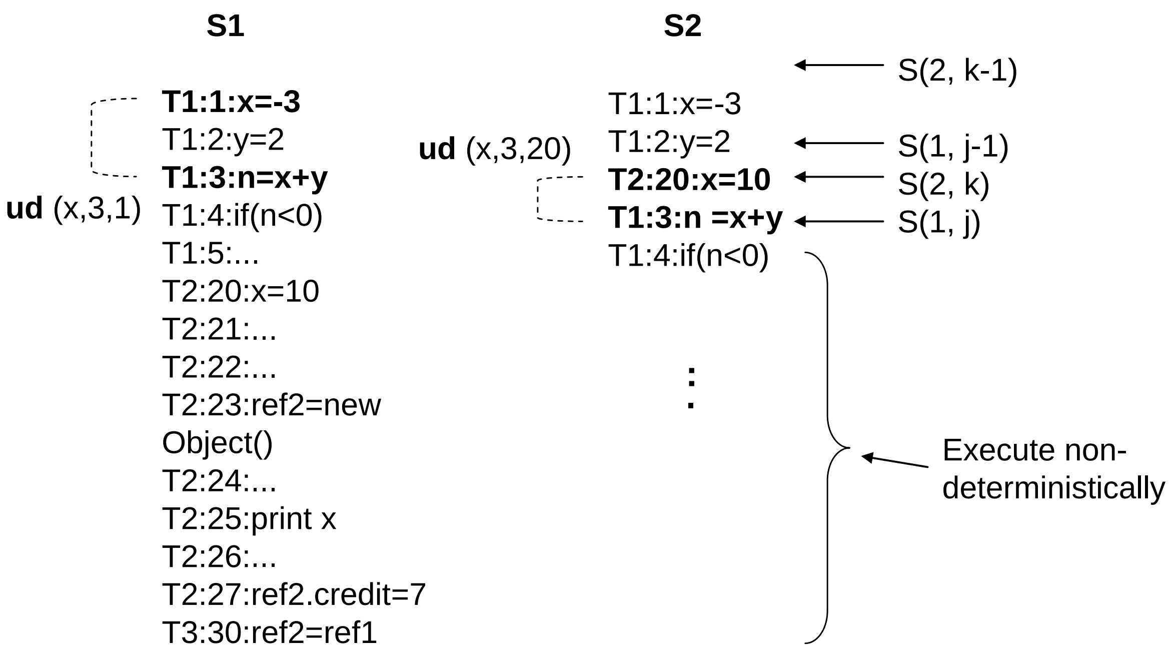


Fig. 9. Example of a test case generation from a “guideline”.

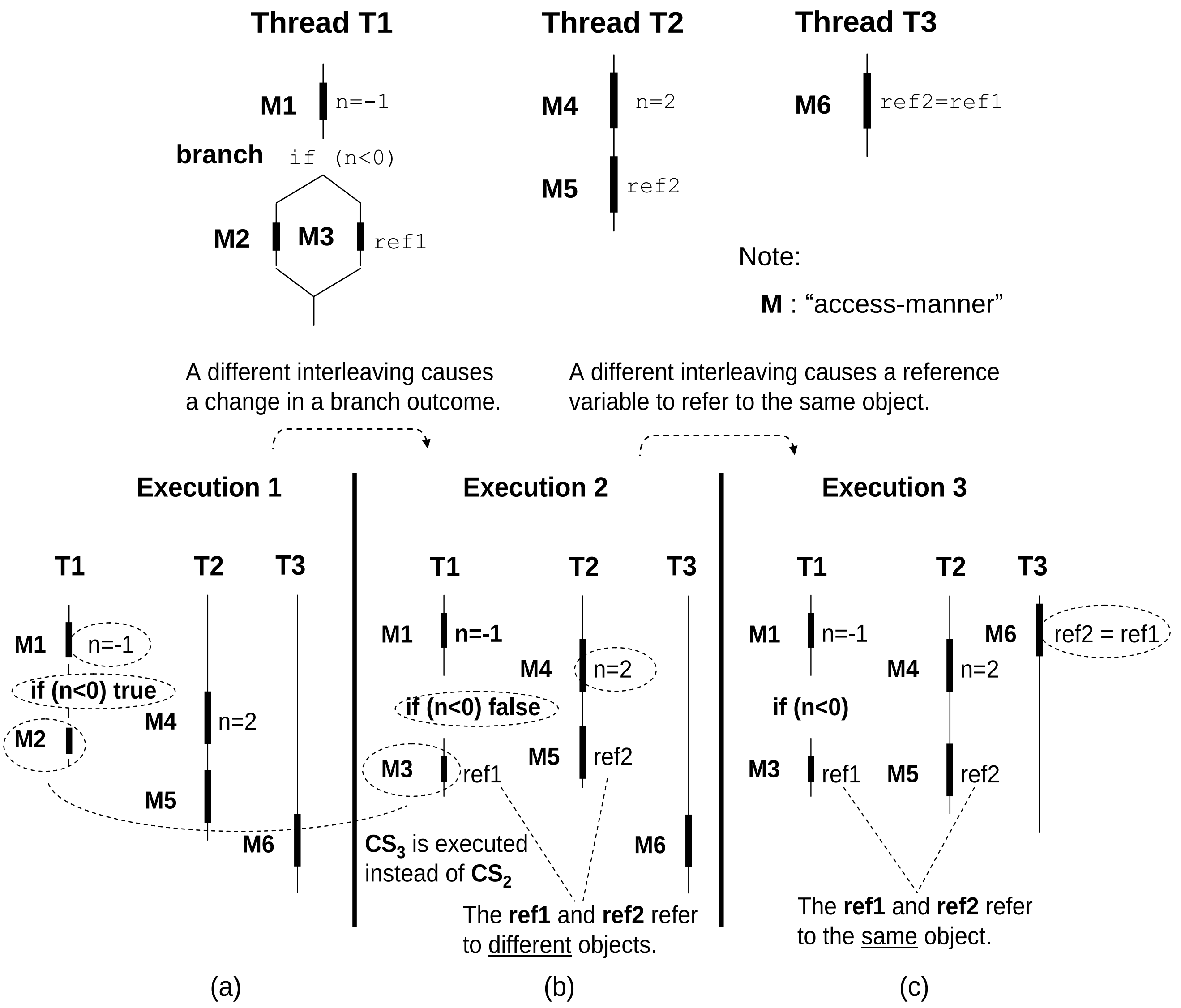

Thread T1
M1 n=-1
branch if (n<0)
M2 M3 ref1
Thread T2
M4 n=2
M5 ref2
Thread T3
M6 ref2=ref1
Note:
M : "access-manner"
A different interleaving causes a change in a branch outcome.
A different interleaving causes a reference variable to refer to the same object.
Execution 1
Execution 2
Execution 3
T1 T2 T3
if (n<0) true
if (n<0) false
CS3 is executed instead of CS2
The ref1 and ref2 refer to different objects.
ref2 = ref1
The ref1 and ref2 refer to the same object.
(a)
(b)
(c)

Fig. 10. Example of three executions with different interleavings.

starts.

Figure 11 is an example of an execution for the source code in Figure 1. It shows some concurrent-pairs of "access-manners". The number of concurrent-pairs of "access-manners" depends on the number of "access-manners" and how they are distributed among threads.

We have to check race conditions for each concurrent-pair of "access-manners". When a "use" operation has more than one member in its *define_set*, its value might be affected by different interleavings. For each concurrent-pair of "access-manners", we have to check race conditions for all the combinations of the *define_set* of the lock variables and reference variables. The occurrence of race conditions might be affected in the event that any lock variables refer to different lock objects or any reference variables refer to different objects. A race condition can occur in Figure 11 between the concurrent-pair of "access-manners" *M1* and *M3*. This happens when the reference variables *ref1* and *ref2* refer to the same object, and the lock variables *a* and *b* refer to different lock objects.

There is no need to check different interleavings between a concurrent-pair of "access-manners" that satisfies the following two conditions, because the consistent locking will be the same:

-- The concurrent-pair of "access-manners" has been checked for race conditions in the previous test execution.

-- Different interleavings will not change the value of lock variables and reference variables.

In this way, we can reduce the number of test cases. On the contrary, if any different interleavings might affect the lock variables or reference variables, then they have to be tested because the consistent locking might be affected accordingly.

Generating Test Cases

This section explains how to generate different interleavings to check whether accesses through reference variables in an "access-manner" have consistent locking.

In Figure 11, the *define_set* for the read operation to *ref2* in *M3* for *pair2* contains two members, hence its value might be affected by different interleavings.

Figure 12 shows an example of a concurrent dependency graph for the "access-manner" *M3* in Figure 11. The root node contains the statements from the "access-manner" *M3*.

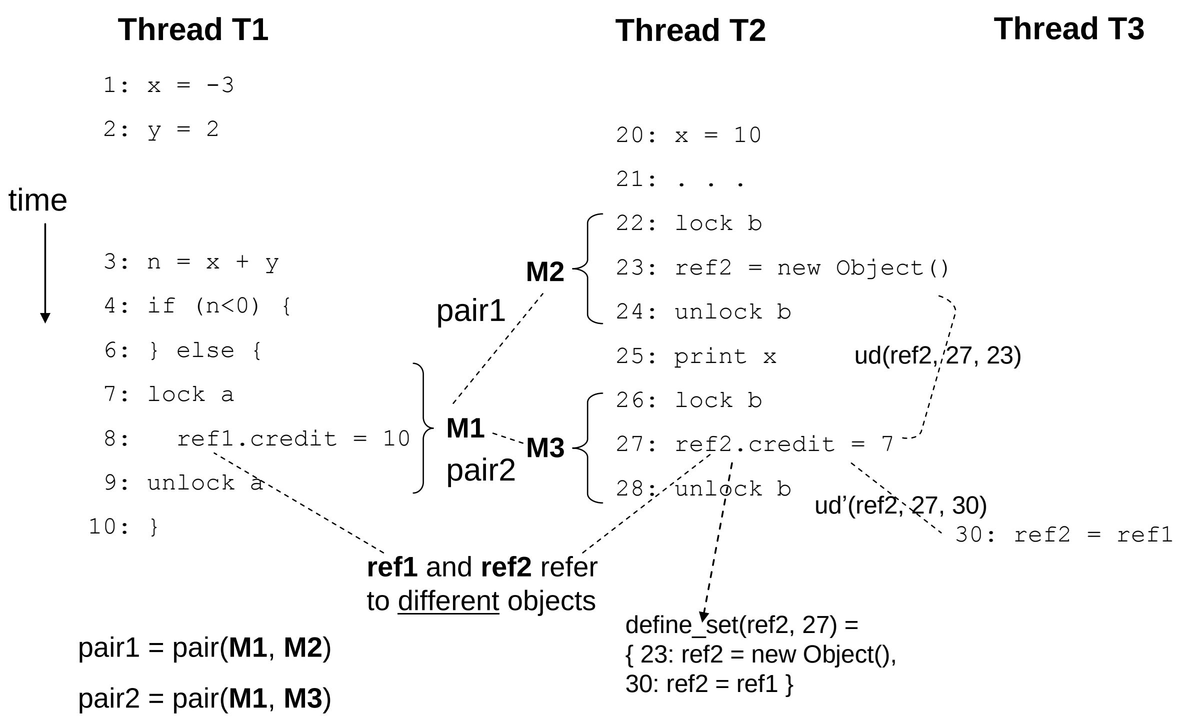


Fig. 11. Example of some concurrent-pairs of "access-manners" in an execution trace.

TABLE V
SET OF "GUIDELINES" FOR GENERATING TEST CASES FOR TESTING PAIR2 IN FIGURE 11

No.	"Guideline"	"Execution-variant"	Test result
1	*gu1* = { *ud*(*ref2*, 27, 23) }	*V1*	No race condition, because *ref1* and *ref2* refer to different objects.
2	*gu2* = { *ud*(*ref2*, 27, 30) }	*V2*	Race condition for accessing *ref1*, if lock *a* and lock *b* refer to different lock objects.

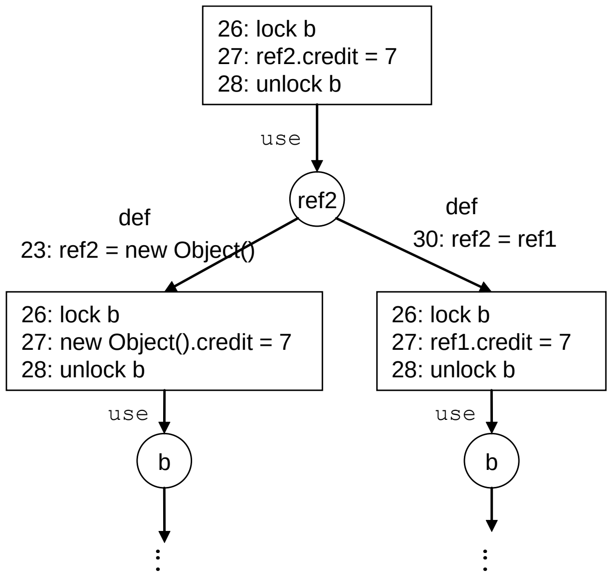


Fig. 12. Example of the concurrent dependency graph for the "access-manner" M3 in Figure 11.

Traversing a Concurrent Dependency Graph of an "Access-Manner"

This subsection shows an example of how to traverse the concurrent dependency graph of the "access-manner" *M3* in Figure 12. Table V shows the results of traversing the concurrent dependency graph in Figure 12 by applying Algorithm 4. Let us assume that the execution in Figure 11 is obtained when the program is first tested, and we call it "execution-variant" *V1*. Its interleaving satisfies the "use-define" *ud*(*ref2*, 27, 23). The "execution-variant" *V2* is used as the next test case as shown in Figure 13. Its interleaving satisfies the "use-define" *ud*(*ref2*, 27, 30). The next subsection explains how to create the "execution-variant" *V2* effectively from the concurrent dependency graph in Figure 12.

Generating Test Cases for Checking Consistent Locking of an "Access-Manner"

Based on Table V, the "execution-variant" *V2* can be generated from "execution-variant" *V1* by changing the "define" operation for the "use" operation of variable *ref2* in line 21.

-- The **"**guideline**"** for the current "execution-variant" *V1*: { *ud*(*ref2*, 27: *ref2.credit* = 7, 23: ref2 = new Object( )) }

-- The **"**guideline**"** for the target "execution-variant" *V2*: { *ud*(*ref2*, 27: *ref2.credit* = 7, 30: *ref2* = *ref1*) }

Generating the "execution-variant" *V2* applies to case 2 because the "use" operation is in a different thread from the target "def" operation. Therefore, Algorithm 6 applies for this case.

-- "def_{base}" : 23: *ref2* = new Object( )
-- "$\text{def}_{\text{target}}$" : 30: *ref2* = *ref1*
-- "use" : 27: *ref2.credit* = 7

Figure 13 shows an example of the execution trace that satisfies the "guideline" *gu2*.

D.Reducing the Effort Involved in Checking Race Conditions

When a new test case is executed, only concurrent-pairs of "access-manners" whose "access-manners" are affected by the new test case have to be re-checked for race conditions. In

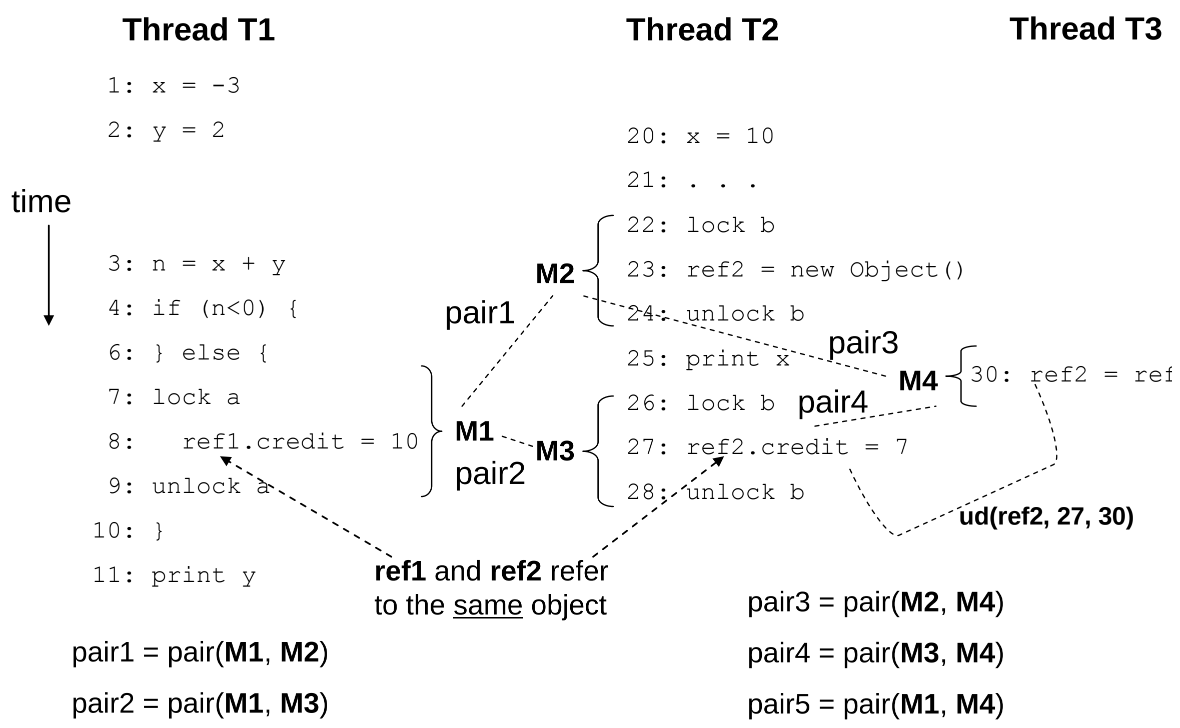


Fig. 13. Example of a test case execution for "execution-variant" V2.

this way, the effort for checking race conditions is reduced. The following discussion explains how to identify the “access-manners” which are affected by a new test case.

Conditional Statements in a Branch

A different interleaving might change branch outcomes which can, in turn, change the sequences of locks and shared variables. In the event that a test case is created based on a conditional statement of a branch, then only the “access-manners” affected by the change of the branch outcomes have to be re-checked for race conditions. Let *op(br, true)* be the set of operations executed only when the conditional statement in a branch *br* is *true* and let *op(M)* be the set of operations within an “access-manner” *M*. When the outcome of the branch *br* changes from *true* to *false*, only check race conditions in concurrent pairs of “access-manners” involving “access-manner” *M*, where $op(br, false) \cap op(M) \neq \emptyset$. Also, when the outcome of branch *br* changes from *false* to become *true*, a similar rule applies. For example, let us assume a test case is created based on the branch in line 4 in Figure 13. If the branch has changed its outcome from *true* to become *false*, then the “access-manner” affected by the test case is *M1*. Therefore, we have to check only those race conditions for the concurrent-pairs related to the “access-manner” *M1*; these are *pair1*, *pair2*, and *pair5*.

Assignment of Lock Variables or Reference Variables within an "Access-Manner"

Different interleavings might change the assignment of lock variables or reference variables within an “access-manner”. If a test case is created based on an “access-manner” M_a, then we have to check only those race conditions for the concurrent pair of “access-manners” *pair(M1, M2)* where $M1 = M_a$ or $M2 = M_a$. The test cases in the example of Table V are created based on the “access-manner” *M3* from Figure 11. Only *pair2* and *pair4* have to be re-checked using a race detector because they are related to the “access-manner” *M3*. On the other hand, since *pair1*, *pair3* and *pair5* are not related to the “access-manner” *M3*, they are not affected by the test case. Hence, there is no need to re-check race conditions among them (see Figure 13).

When a loop contains an “access-manner”, each iteration can generate a concurrent-pair of “access-manners”. In the case of an infinite loop, the number of concurrent-pairs of “access-manners” can be infinite. However, in some cases the concurrent-pairs generated in each iteration could be the same as in the previous one. In such cases, there is no need to check for all the iterations. In this way, the effort involved in checking race conditions during the test can be reduced. We will show an example of this in subsection C of section IV on experiments.

IV. Experiments

In this section, we show the effectiveness of our proposed new method in reducing the memory required for generating test cases. The work by T. E. Setiadi, A. Ohsuga, and M. Maekawa [23] requires a variant graph from the existing reachability testing method. The effectiveness of our proposed new method is demonstrated by comparing the size of our proposed concurrent dependency graph against that of the variant graph. We discuss three experiments using the following multi-threaded Java open source programs [23]:

1. jNetMap [25] is a network client for monitoring devices, such as PCs and routers, in a network.
2. Apache Commons Pool [26] is a generic object-pooling library from Apache.
3. Jobo [27] is a web spider for downloading complete websites to a local computer.

Table VI shows that the concurrent dependency graph proposed in this paper is smaller in size than the variant graph in the existing reachability testing method.

TABLE VI
COMPARISON OF THE EXISTING VARIANT GRAPH AND THE PROPOSED CONCURRENT DEPENDENCY GRAPH

NO	Programs	Number of nodes	
		Existing variant graph (from our previous work using the reachability testing method)	Concurrent dependency graph (from our proposed new method)
1	jNetMap	Infinite	8
2	Apache Commons Pool	990	4
3	Jobo	Infinite	4

A. Experiment 1: jNetMap

There is an access to a shared variable in an infinite loop affected by another thread. This causes an infinite sequence of read-write operations and creates a variant graph of infinite size. Figure 16 shows only some parts of the variant graph from the reachability testing method. Here we explain only one example that caused a redundancy.

Figure 14 shows the execution trace of the first execution. The reachability testing method considers all different interleavings between the two threads that can affect the values of shared variables. On the other hand, our proposed method considers only different interleavings that can possibly change the outcome of the conditional statement in line 279, so it generates fewer test cases. In this experiment, only the conditional statement in line 279 might cause different sequences of locks and shared variables.

Figure 15 shows a concurrent dependency graph for the branch from the execution trace analysis of the first execution. The traversal of the concurrent dependency graph in Figure 15 results in a set of “guidelines” in Table VII for generating test cases. Table VII shows the set of “guidelines” for producing two test cases based on the traversal of the concurrent dependency graph in Figure 15.

TABLE VII
SET OF “GUIDELINES” FROM THE CONCURRENT DEPENDENCY GRAPH IN FIGURE 15

NO.	“Guideline”
1	*gu1* = { *ud*(*pingInterval*, 279, 69) }
2	*gu2* = { *ud*(*pingInterval*, 279, 112) }

The branch outcomes for the conditional statement in line 279 are determined by the assignment from the write operation in either line 69 or 112. For a comparison with the existing reachability testing method, we created a variant graph in Figure 16 based on the execution trace in Figure 14.

We refer to the source code in Figure 17 to explain the cause of redundancy. The truth value of the branch in line 279 is affected by the order of interleavings between the assignment of shared variable *pingInterval* in line 69 and 112. The other read and write operations to the shared variable *pingInterval* in line 123, 284, and 286 do not affect the truth value of the branch in line 279, so different interleavings among them are redundant. For exploring different execution paths caused by the branch in line 279, we have to consider only whether an “execution-variant” satisfies the *ud*(*pingInterval*, 279, 69) or *ud*(*pingInterval*, 279, 112). In other words, we can group those “execution-variants” into two groups and it is sufficient to test only one of each group.

B. Experiment 2: Apache Commons Pool

The reachability testing method uses a variant graph with 990 nodes to generate 216 test cases. However, most of them do not affect the occurrence of the race condition. As shown in the work by T. E. Setiadi, A. Ohsuga, and M. Maekawa [23], only two test cases are actually required. Figure 19 shows that we require a concurrent dependency graph with only 4 nodes to generate those two required test cases.

Figure 18 shows the execution trace of the test program containing race conditions. The reachability testing method considers all different interleavings that affect the values of shared variables among the three threads in Figure 18. Our proposed method generates fewer test cases because it considers only those interleavings that can possibly affect the conditional statement in line 906. Figure 19 shows a concurrent dependency graph from the execution trace in Figure 18.

TABLE VIII
SET OF “GUIDELINES” FROM THE CONCURRENT DEPENDENCY GRAPH IN FIGURE 19

NO.	“Guideline”
1	*gu1* = { *ud*(*_numActive*, 906, 126) }
2	*gu2* = { *ud*(*_numActive*, 906, 765) }

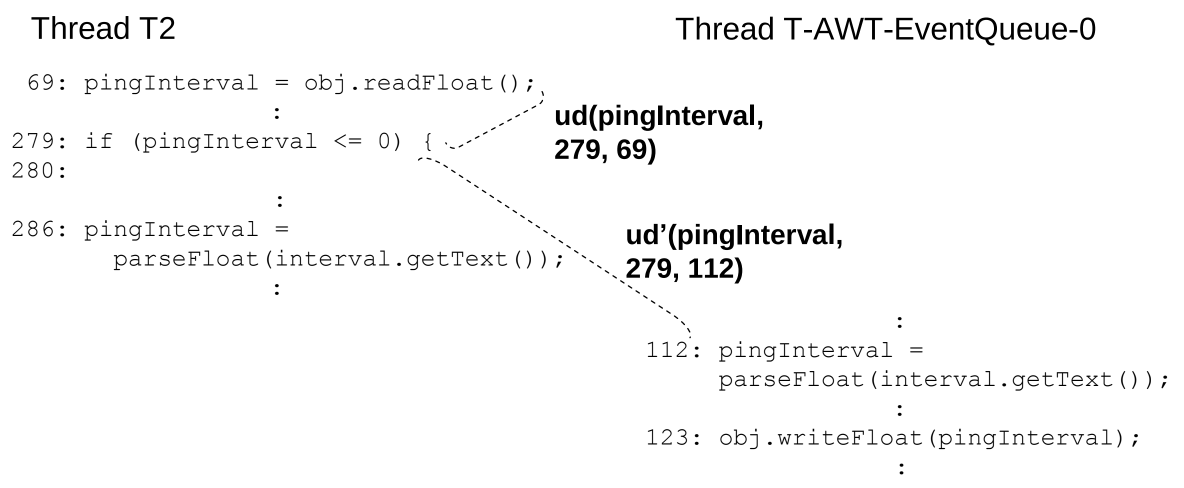
Based on the set of “guidelines” in Table VIII, our proposed method generates only 2 test cases. Figure 20 shows a piece of code to explain the cause of redundancy in the reachability testing method. The conditional statement in line 906 depends only on the values of the shared variable *_numActive* affected by the interleavings with the assignment in line 765 of the thread *T3*. The access through the reference variable *_pool* depends on interleavings, but it does not affect the conditional statement in line 906. Hence, different interleavings that are affecting the reference variable *_pool* are redundant. Figure 21 shows the concurrent dependency graph for the reference variable *_pool*.

C. Experiment 3: JoBo

In this experiment, we downloaded a website from Yahoo [28] and saved it in a local computer. Similar to Experiment 1, there is an access to a shared variable within an infinite loop. This shared variable is affected by another thread, thus causing an execution trace of infinite length accessing the shared variable in question.

The reachability testing method produces a variant graph of infinite length because of this execution trace of infinite length. As shown in the work by T. E. Setiadi, A. Ohsuga, and M. Maekawa [23], only two test cases are required. Figure 23 shows the concurrent dependency graph for creating the two required test cases.

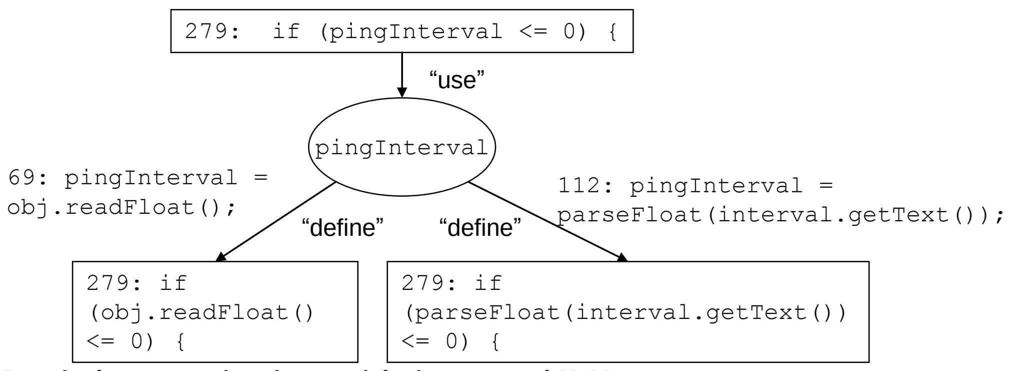
Figure 22 shows the execution trace of the first execution. Note that loop 1 is an infinite loop. The infinite loop in the thread *T3* is accessing a shared variable. For each access to a shared variable in the loop iteration, its value can be affected by the assignment from the thread *T4*. Therefore, the reachability testing method generates infinite test cases because it produces a different test case for each iteration in


Thread T2
Thread T-AWT-EventQueue-0
ud(pingInterval, 279, 69)
ud'(pingInterval, 279, 112)

```
 69: pingInterval = obj.readFloat();
                  :
279: if (pingInterval <= 0) {
280:
                  :
286: pingInterval =
       parseFloat(interval.getText());
                  :
```

```
                  :
112: pingInterval =
     parseFloat(interval.getText());
                  :
123: obj.writeFloat(pingInterval);
                  :
```

Fig. 14. Execution trace of the first test execution of jNetMap.



```
279:  if (pingInterval <= 0) {
```

```
69: pingInterval =
obj.readFloat();
```

```
112: pingInterval =
parseFloat(interval.getText());
```

```
279: if
(obj.readFloat()
<= 0) {
```

```
279: if
(parseFloat(interval.getText())
<= 0) {
```

Fig. 15. Example of a concurrent dependency graph for the execution of jNetMap.

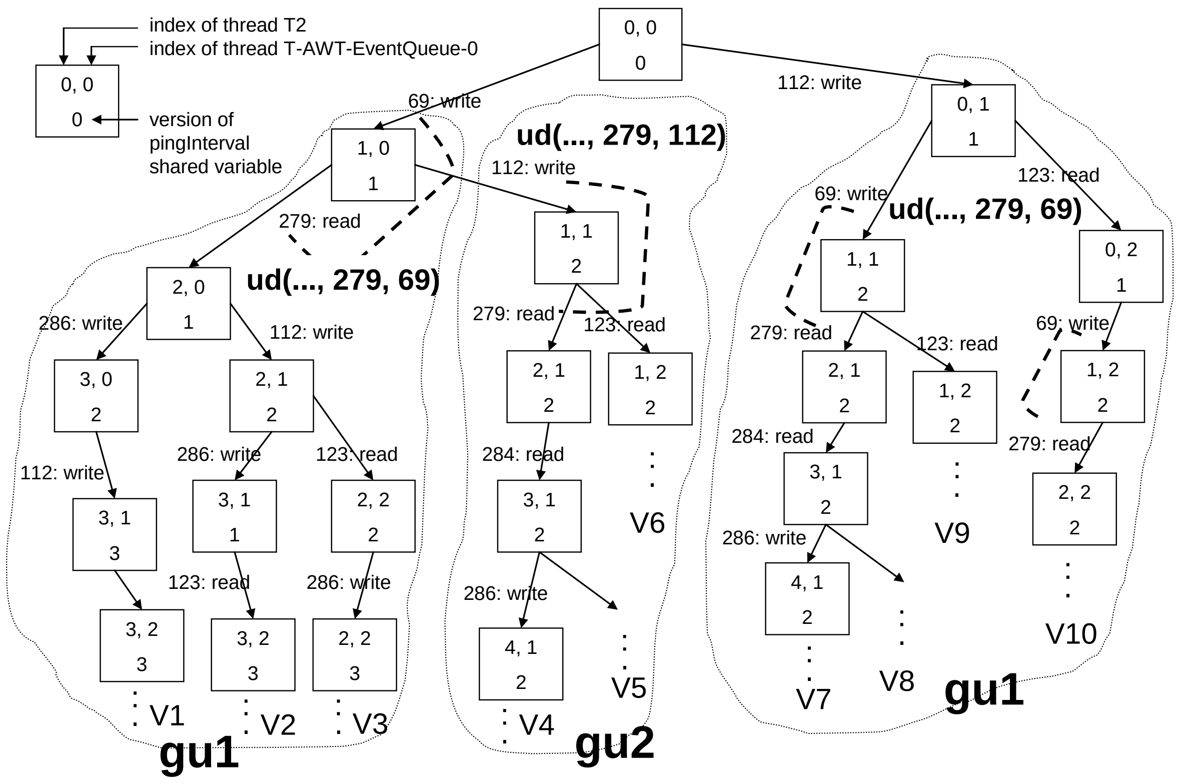
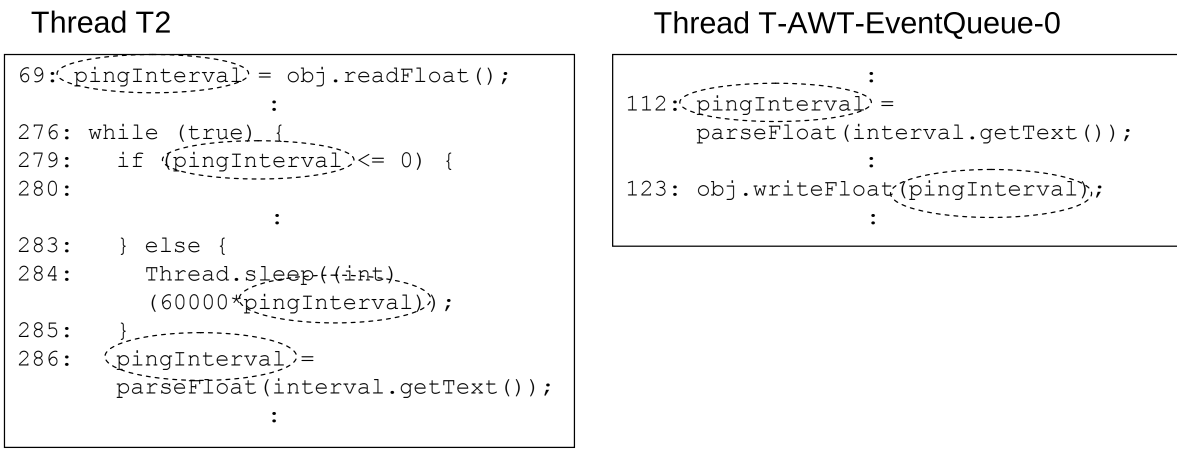


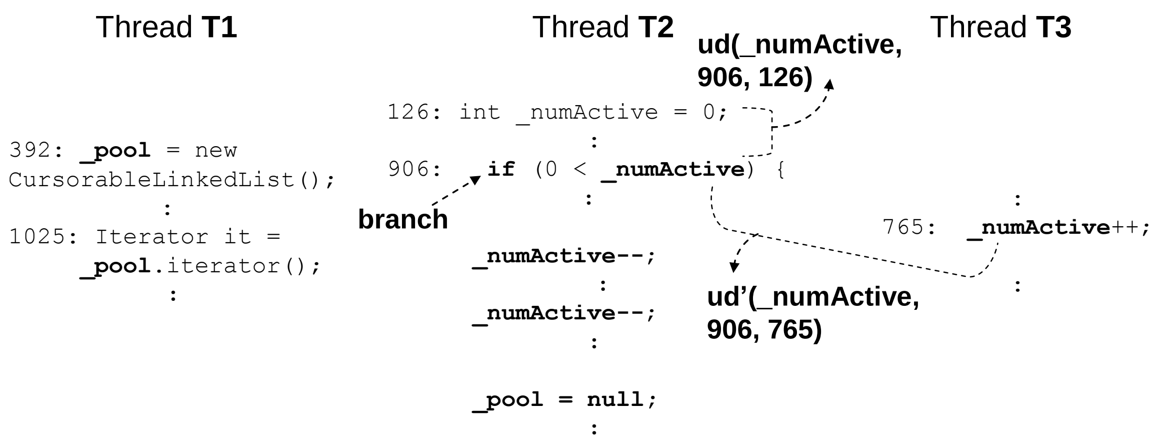
Fig. 16. Variant graph for the execution of jNetMap.



```
69: pingInterval = obj.readFloat();
                 :
276: while (true) {
279:   if (pingInterval <= 0) {
280:
                 :
283:   } else {
284:     Thread.sleep((int)
         (60000*pingInterval));
285:   }
286:   pingInterval =
       parseFloat(interval.getText());
                 :
```

```
                 :
112: pingInterval =
     parseFloat(interval.getText());
                 :
123: obj.writeFloat(pingInterval);
                 :
```

Fig. 17. The source code of jNetMap.



```
392: _pool = new
CursorableLinkedList();
          :
1025: Iterator it =
      _pool.iterator();
          :
```

```
126: int _numActive = 0;
          :
906:   if (0 < _numActive) {
          :
  _numActive--;
          :
  _numActive--;
          :

  _pool = null;
          :
```

```
          :
765:  _numActive++;
          :
```

Fig. 18. Execution trace of the experiment using Apache Commons Pool.

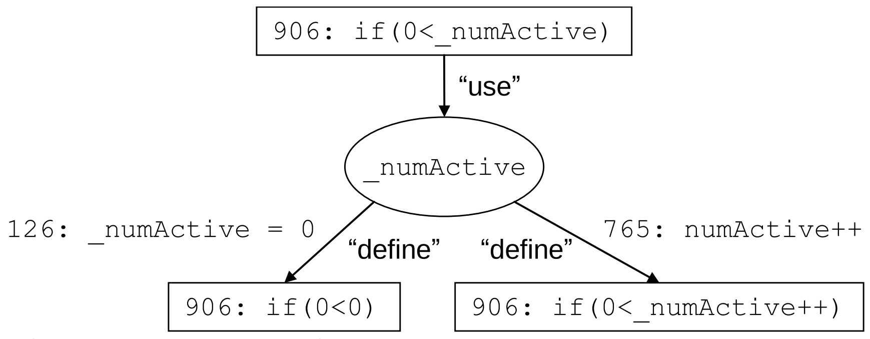


Fig. 19. Example of a concurrent dependency graph for Apache Commons Pool.

the infinite loop. Our method identifies that only some of the iterations are sufficient for checking consistent locking, because the concurrent-pair of "access-manners" generated for each iteration is the same as in the previous one

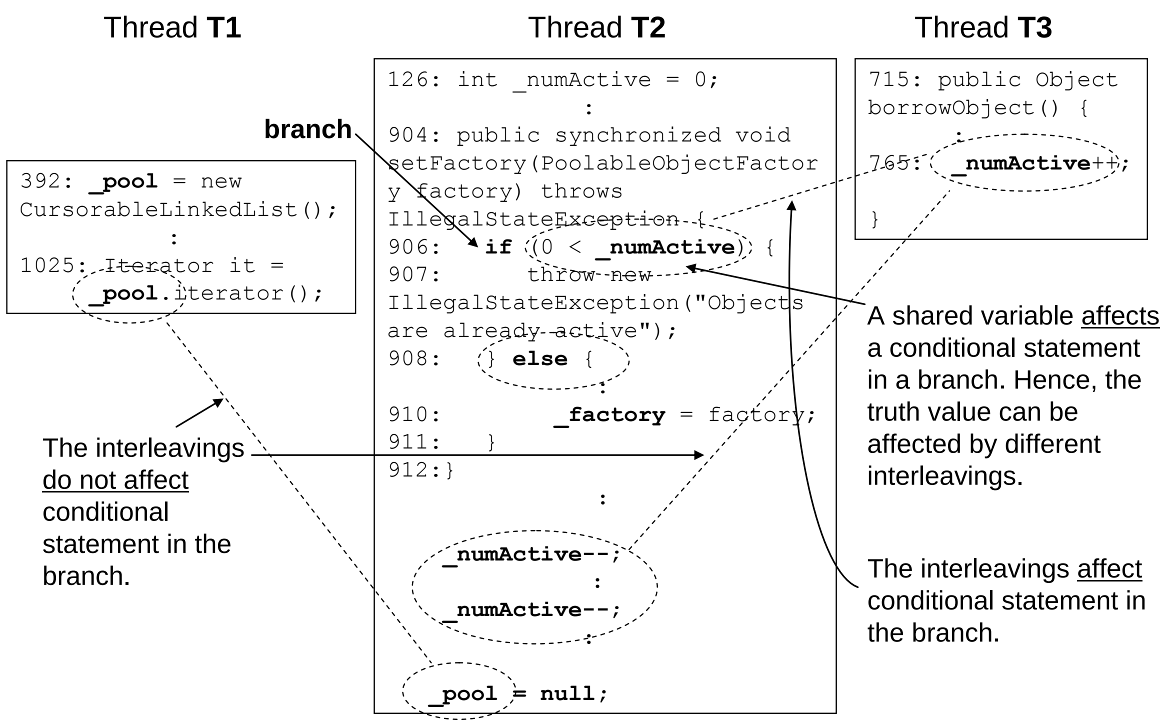
Figure 23 shows a concurrent dependency graph for the branch from the execution trace analysis of the first test execution in Figure 22. Based on the traversals of the concurrent dependency graph in Figure 23, our proposed method produces the set of "guidelines" in Table IX. We then generate two test cases based on Table IX.

Figure 24 shows the piece of code that affects the test case generation. There is an infinite loop in the thread *T3* accessing a shared variable. From the execution trace of the first execution, the reachability testing method produces a variant graph with infinite nodes. For each node, an "execution-variant" can be created by making a different order of interleavings for an assignment from the thread *T4*, hence causing an infinite number of test cases.

TABLE IX
SET OF "GUIDELINES" FROM THE CONCURRENT DEPENDENCY GRAPH IN FIGURE 23

NO.	"Guideline"
1	*gu1* = { *ud*(*m_connection*, 43, 9) }
2	*gu2* = { *ud*(*m_connection*, 43, 145) }

The first and second loop iterations of the execution trace in Figure 22 satisfy the first "use-define" in the "guideline" *gu1*, whereas the third iteration satisfies the second "use-define" in the "guideline" *gu2*. The first iteration of the infinite loop has the same concurrent-pair of "access-manners" as the second iteration, whereas the third.


Thread **T1**
Thread **T2**
Thread **T3**
branch
The interleavings do not affect conditional statement in the branch.
A shared variable affects a conditional statement in a branch. Hence, the truth value can be affected by different interleavings.
The interleavings affect conditional statement in the branch.

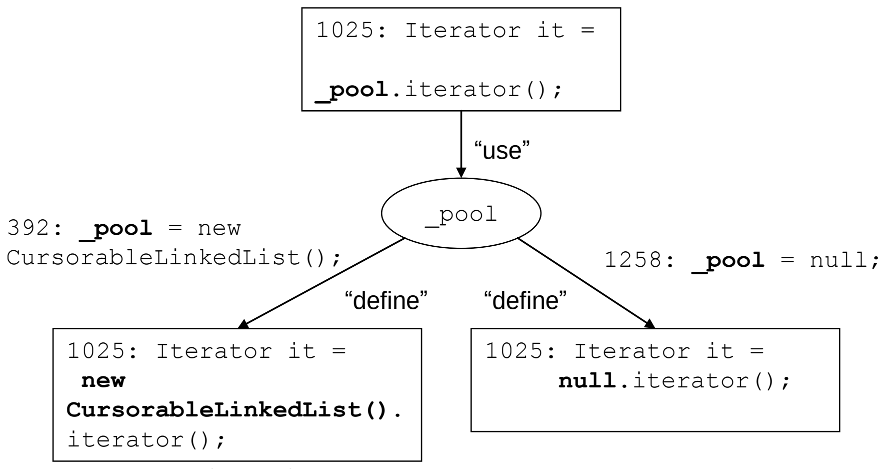
```
392: _pool = new
CursorableLinkedList();
        :
1025: Iterator it =
    _pool.iterator();
```

```
126: int _numActive = 0;
             :
904: public synchronized void
setFactory(PoolableObjectFactor
y factory) throws
IllegalStateException {
906:   if (0 < _numActive) {
907:      throw new
IllegalStateException("Objects
are already active");
908:   } else {
             :
910:         _factory = factory;
911:   }
912:}
             :
  _numActive--;
             :
  _numActive--;
             :
  _pool = null;
```

```
715: public Object
borrowObject() {
    :
765:  _numActive++;

}
```

Fig. 20. Example of a test program using the Apache Commons Pool library.



```
1025: Iterator it =

_pool.iterator();
```

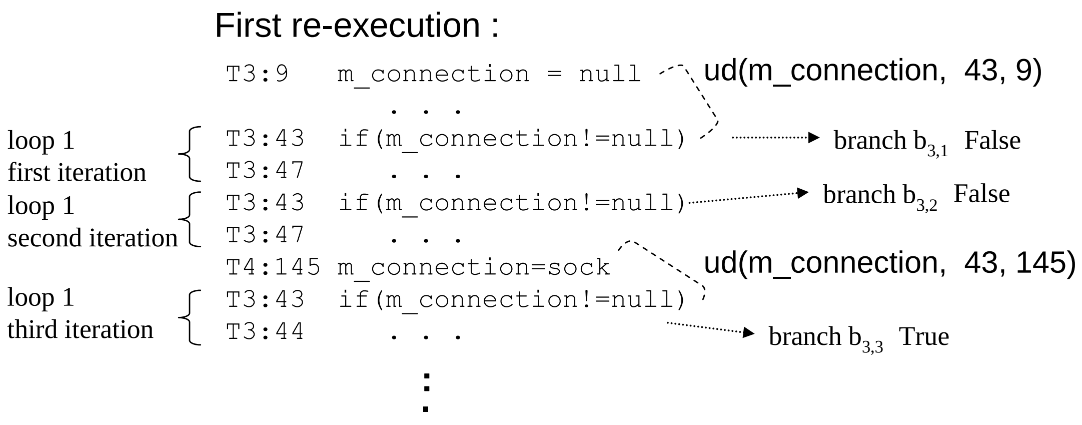
```
392: _pool = new
CursorableLinkedList();
```

```
1258: _pool = null;
```

```
1025: Iterator it =
 new
CursorableLinkedList().
iterator();
```

```
1025: Iterator it =
     null.iterator();
```

Fig. 21. Concurrent dependency graph for the reference variable *_pool*.



```
T3:9    m_connection = null
            . . .
T3:43   if(m_connection!=null)
T3:47       . . .
T3:43   if(m_connection!=null)
T3:47       . . .
T4:145  m_connection=sock
T3:43   if(m_connection!=null)
T3:44       . . .
              :
```

Fig. 22. Execution trace of the first test.

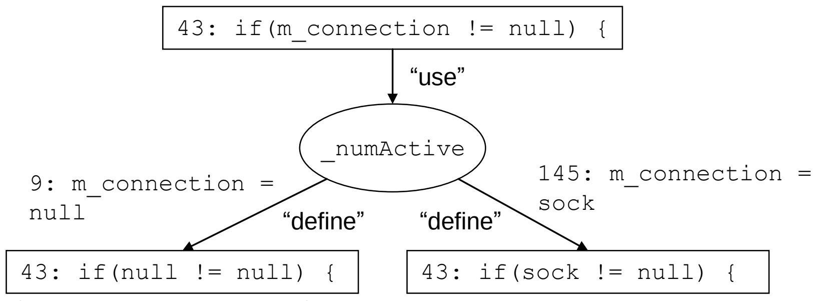
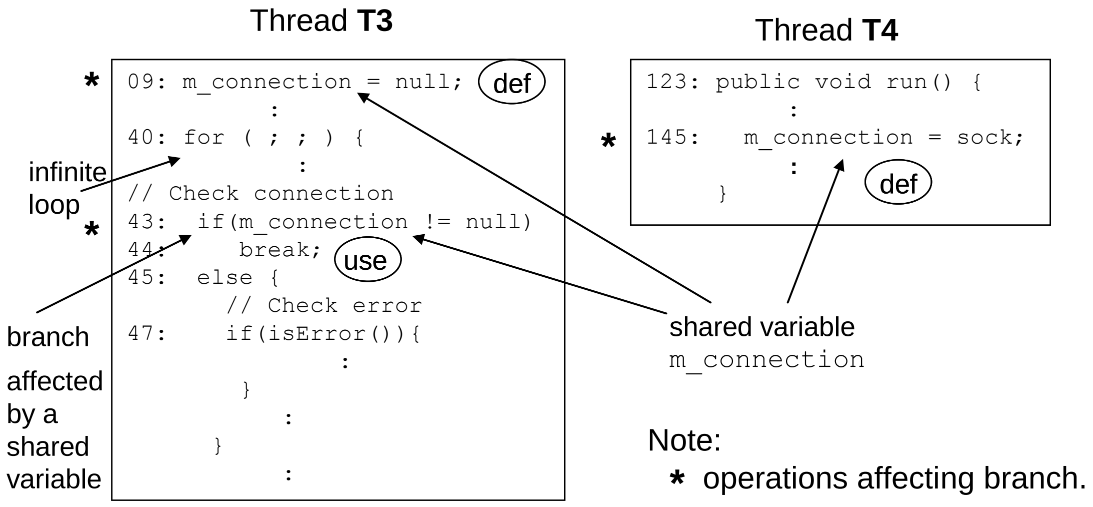


Fig. 23. Example of a concurrent dependency graph for JoBo.


Thread T3
Thread T4
infinite loop
branch affected by a shared variable
def
use
def
shared variable m_connection
Note:
* operations affecting branch.

```
* 09: m_connection = null;
          :
  40: for ( ; ; ) {
          :
  // Check connection
* 43:  if(m_connection != null)
  44:     break;
  45:  else {
         // Check error
  47:    if(isError()){
               :
        }
          :
     }
          :
```

```
  123: public void run() {
          :
* 145:   m_connection = sock;
          :
      }
```

Fig. 24. The source code of JoBo.

one has a different concurrent-pair of "access-manners". All possible different concurrent-pairs of "access-manners" in the iterations of loop 1 have been explored, from the first iteration until the third one in the first execution. Therefore our proposed method does not need to test all the infinite loop iterations, because the remaining loop iterations will not produce different sequences of locks and shared variables.

V. Discussions and Future Work

The proposed method applies for concurrent programs that are using lock mechanisms. It is effective in reducing the number of test cases by considering only different interleavings that are affecting race conditions. Our method also supports file references, such as file_name = fopen(c:\\data\...), because we can treat them in a similar way to reference variables.

Even though a lock variable can refer to different lock objects or a reference variable can refer to different objects, there are some special conditions in which we can guarantee there will be no race conditions caused by different interleavings among concurrent-pairs of "access-manners" *M1* and *M2*. In such a situation, the test cases can be further reduced because there is no need to check the interleavings between them. This can happen when no same object is referred to in either *M1* or *M2* even though different interleavings are applied between them. Formally, we define this as follows:
Let *vars(M)* be the set of reference variables within an "access-manner" *M*. There is no intersection between objects referred to by *vars(M1)* and *vars(M2)*.

VI. Conclusion

We proposed an efficient new method to generate different interleavings as test cases for detecting race conditions. This represents a refinement of the existing method for test case generation and dynamic race detection. We expanded on past work, [24], [29] and [30], concerning reachability testing. The original aspects of our proposed method are as follows:

-- Utilizing data dependency by proposing concurrent dependency graphs to guide the test case generation, in order to generate only different interleavings that might affect consistent locking. In this way, we can avoid generating redundant test cases.

-- Generating test cases for checking the consistent locking of accesses through reference variables. We show that our method to generate test cases for detecting race conditions caused by branching can also be applied for detecting race conditions caused by accesses through reference variables.

-- Dividing an execution trace into several concurrent-pairs of "access-manners" to reduce the effort involved in checking race conditions. Race conditions have to be re-checked only among concurrent-pairs of "access-manners" that are affected by new test cases. When a test case changes branch outcomes, our proposed method checks whether any of the "access-manners" are affected by these branch outcomes. Subsequently, only the concurrent-pairs related to those "access-manners" that are affected by the change of branch outcomes have to be re-checked. Similarly, when a test case changes a lock variable or a variable reference within any "access-manners", only the concurrent-pairs related to the affected "access-manners" have to be re-checked. In this way, we can reduce the effort involved in checking race conditions during the tests.

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] C. Praun and T. Gross, "Object race detection," in *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pp. 70–82, 2001.

[3] M. Christiaens and K. De. Bosschere, "TRaDe, a topological approach to on-the-fly race detection in Java programs," in *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM)*, April 2001.

[4] 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.

[5] 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, pp. 93-110, February 2006.

[6] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: efficient detection of data race conditions via adaptive tracking," *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[7] J.D. Choi, B.P. Miller, and R.H.B Netzer, "Techniques for debugging parallel programs with flowback analysis," *ACM Transactions on Programming Languages and Systems*, 13(4), pp. 491–530, 1991.

[8] 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.

[9] C. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient Data Race Detection for Distributed Memory Parallel Programs," *SC11*, Seattle, Washington, USA Copyright 2011 ACM 978-1-4503-0771-0/11/11, November 12-18, 2011.

[10] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur, "Testing multi-threaded Java programs," *IBM System Journal, Special Issue on Software Testing*, February 2002.

[11] 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*, 15(3-5), pp. 485-499, 2003.

[12] M. Musuvathi,S. Qadeer, and T. Ball, "CHESS: A systematic testing tool for concurrent software," *Microsoft Research Technical Report*, MSR-TR-2007-149, 2007.

[13] 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.

[14] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *CAV. Springer*, pp. 419-423, 2006.

[15] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria." in *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software*, 2007.

[16] C.D. Yang and L.L. Pollock, "All-uses testing of shared memory parallel programs," *Software Testing, Verification, and Reliability (SVTR) Journal*, vol. 13, no. 1, pp. 3-24, 2003.

[17] C. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path coverage for parallel programs," In: *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, Clearwater Beach, Florida, United States. ISBN:0-89791-971-8, pp. 153-162, 1998.

[18] H. Kojima, Y. Kakuda, J. Takahashi, and T. Ohta, "A Model for Concurrent States and Its Coverage Criteria," *International Symposium on Autonomous Decentralized Systems*, ISADS '09, pp. 23-25, 2009.

[19] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural Testing of Concurrent Programs," *IEEE Trans. Soft. Eng*, vol.18, no.3, pp. 206-215, 1992.

[20] J. Takahashi, H. Kojima, and Z. Furukawa, "Coverage Based Testing for Concurrent Software," *The 28th IEEE International Conference on Distributed Computing Systems Workshops*, 533-538, 2008.

[21] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of Synchronization Coverage," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, Chicago, IL, USA. ISBN:1-59593-080-9, pp. 206-212, 2005.

[22] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka, "Testing concurrent programs: a formal evaluation of coverage criteria," *Seventh Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE '96)*, Herzliya, ISRAEL. ISBN: 0-8186-7536-5, 1996.

[23] T. E. Setiadi, A. Ohsuga, and M. Maekawa, "Efficient Execution Path Exploration for Detecting Races in Concurrent Programs," *IAENG International Journal of Computer Science*, vol. 40, issue 3, pp. 143 – 163, September 2013.

[24] G. H. Hwang, K. C. Tai, and T. L. Huang, "Reachability Testing: An approach to testing concurrent software," *International Journal of Software Engineering and Knowledge Engineering*, 1995.

[25] jNetMap, June 2009. Available: http://www.rakudave.ch/?q=jnetmap

[26] Apache Commons Pool, 2006. Available at: http://jakarta.apache.org/commons/pool/

[27] D. Matuschek, JoBo: web spider, Dec 2006. Available at: http://www.matuschek.net/jobo-menu/

[28] Yahoo. Available at: http://www.yahoo.com

[29] R. Carver and Y. Lei, "A general model for reachability testing of concurrent programs," *International Conference on Formal Engineering Methods*, pp. 76-98, 2004.

[30] Y. Lei, and R. H. Carver, "Reachability testing of concurrent programs," *IEEE Transactions on Software Engineering*, vol. 32, issue 6, pp. 382- 403, 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.