Automated Testing of Definition-Use Data Flow for Multithreaded Programs

Xiaodong Zhang*, Zijiang Yang†, Qinghua Zheng*, Pei Liu*, Jialiang Chang‡, Yu Hao* and Ting Liu*,
*Ministry of Education Key Lab for Intelligent Networks and Network Security,
Xi’an Jiaotong University, Xi’an, Shaanxi 710000, China
Email: {xdzhang.pliu,yhao}@sei.xjtu.edu.cn, {qhzheng, tingliu}@mail.xjtu.edu.cn
†Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA,
Email: {zijiang.yang,jialiang.chang}@wmich.edu

Abstract—With the advent of multicore processors, there is a trend towards multithreading to take advantage of parallel computing resources. Due to greatly increased complexity, programmers need effective testing methodology that can thoroughly test multithreaded programs. There has been significant progress based on symbolic execution that attempts to exhaustively explore all the intra-thread paths and inter-thread interleavings. However, such testing approach faces two insuperable challenges. Firstly, exploring an astronomically large number of paths and interleavings limits its scalability. Secondly, a path itself does not directly help programmers understand program behavior. In this paper, we propose an alternate testing methodology that focuses on definition-use data flow instead of paths/interleavings. Such approach not only leads to orders of magnitude reduction in testing complexity, but also gives programmers direct help on examining the shared variable usage in a multithreaded program.

Index Terms—Multithreaded Program; Symbolic Analysis; Guided Execution; Definition-Use

I. INTRODUCTION

Concurrent programming is a key technique to unleash the full potential of present and future generations of parallel computing systems based on multi-core processors. However, the intrinsic nondeterminism of parallel execution can result in concurrency errors that are difficult to detect, reproduce, and debug. While most mainstream programming languages today support concurrency, their testing tools are designed and optimized primarily for sequential software development. To address this issue, recent work exploits symbolic-based analysis to handle the challenges unique to multithreaded programs.

Symbolic execution [1–6] has recently regained prominence as a technique for various software engineering tasks [7, 8]. It uses symbolic inputs instead of concrete inputs as the program inputs to drive program execution. Through encoding the path condition as a quantifier-free, first-order logic formula and then deciding the formula with a constraint solver, symbolic execution can systematically explore the paths of a sequential program and generate the corresponding test inputs. Inspired by the success, recent research [9, 10] extends the approach to systematically explore the intra-thread paths and inter-thread interleavings of multithreaded programs. Indeed, the capability of exhaustive coverage can detect subtle bugs that evade ad-hoc testing approaches.

However, even for sequential programs, exhaustive path coverage is often not achievable due to the inherent path explosion problem. That is, the number of feasible paths in a program usually grows exponentially with the increase of program size. Even for a medium-size program, systematically exploring the paths is prohibitively expensive. The scalability issue is exaggerated by multithreading, where exhaustive path and interleaving coverage leads to even more intractable double-exponential growth. Although various heuristics have been proposed, double-exponential growth remains an insuperable challenge for exhaustive path/interleaving coverage.

Besides poor scalability, the other reason preventing its adoption is that programmers do not directly gain insight about a concurrent program through path/interleaving coverage. Unless a path triggers a bug such as assertion failure, a programmer rarely examines any explored path and thus misses the opportunity to detect hidden issues not under scrutiny. This is because the goal of path/interleaving coverage is to execute as many paths as possible. Detection of failures becomes a side-effect during the exploration. Consider the code snippet shown in Figure 1 with three threads and two branches per thread. A complete traversal of the paths and interleavings does not reveal any visible failures. Except gaining certain assurance, a programmer’s knowledge about the program does not change after the testing.

For a multithreaded program, different threads share information via reading and writing of shared variables. This is the reason that makes multithreaded program hard to comprehend and the source of many concurrency bugs [11]. Although concurrency bugs may appear less frequently than sequential bugs, they can cause more severe consequences, such as data corruption, hanging systems, or even catastrophic disasters. Many of the concurrency errors share a common characteristic: when triggered, they usually are followed by an incorrect data flow, i.e., a read instruction uses the value from an unexpected definition. This type of errors is categorized as order violations. According to a study on real-world concurrency bug characteristics [12], order violations account for one third of all non-deadlock concurrency bugs. However, unlike data races [13–15] and atomicity violations [16–19] that receive plenty attention, order violation bugs have been neglected [11]. According to [11], a program is likely to have a bug if
the following three definition-use (DefUse) invariants are not maintained.

- Local/Remote invariants. A read only uses definitions from either the local thread or a remote thread.
- Follower invariants. When there are two consecutive reads upon the same variable from the same thread, the second always uses the same definition as the first.
- Definition set invariants. A read should always use definition(s) from a certain set of writes.

Although a violation of the three invariants does not necessarily mean a bug, we do believe that it warrants careful examination when shared variables use different definitions in different runs. For example, although a complete path/interleaving traversal of the program in Figure 1 does not produce any visible bugs such as assertion failures, our approach shows inconsistent DefUse relations. Therefore, we believe our testing methodology not only detects order violations but also gives programmers insight on how data are communicated between threads.

![Fig. 1. A multithreaded program with three threads. The initial value of the shared variable is }x=1{, and }a,b,c{ are local input variables.](image)

In this paper, we propose a testing methodology that targets DefUse data flow. Our testing approach encodes an execution trace into first-order logic formulas and find new DefUse relations by leveraging an SMT solver to solve the formulas. Then, it explores new paths that may contain unexplored DefUse pairs. The goal of our DefUse testing is to find all the DefUse pairs for shared variables. Such goal makes DefUse testing orders of magnitude cheaper than path/interleaving testing due to the following two reasons: (1) it considers instruction combination between pair-wise threads rather than among all the threads, and (2) each statement within a thread may be considered in isolation rather than an enumeration of all the paths.

Considering the interleaving among the three threads in Figure 1, the number of paths is 64 \((4 \times 4 \times 4)\). If we want to enumerate all the possible DefUse relation, we need to cover all the interleavings between the three pairs of the threads \(T_0 \parallel T_1, T_1 \parallel T_2, T_2 \parallel T_0\). The number of paths between a pair is 16 \((4 \times 4)\), and the total number of paths to enumerate all the DefUse is 48 \((16 \times 3)\). If we extend the example to \(N\) threads with \(M\) if-then-else branches per thread, a complete interleaving/path coverage requires a traversal of \(2^{MN}\) paths, while a complete DefUse coverage needs exploring \(C_N^2 \times (2M \times 2M)\) paths. This is a reduction from \(O(2^{MN})\) to \(O((MN)^2)\). Of course, most programs have more complicated structure than the trivial program shown in Figure 1. For a particular DefUse pair to manifest, it may require cooperation from other statements within the two threads as well as careful synchronization with other threads.

In summary, this paper makes the following contributions.

1) We propose a new testing methodology for multithreaded programs. By exhausting all possible DefUse relations, it offers insight about multithreading at a much lower cost than path/interleaving coverage.

2) We develop algorithms that enables automated systematic DefUse coverage for multithreaded programs.

3) We have implemented a tool called STEM (Systematic Testing of DefUse for Multithreaded programs) and conducted experiments.

The rest of the paper is organized as follows. Relevant terms used in this paper are defined and explained in Section II, followed by a formal presentation of algorithms in Section III. The experimental results are reported in Section IV. Section V describes the threats to validity. Section VI reviews the related work and Section VII concludes the paper.

II. DEFINITIONS

In this section we define and explain relevant terms used in the description of the DefUse testing algorithms.

- **DefUse pair**: An assignment statement is a definition of variable \(v\) if \(v\) is on the left-hand-side of the statement. If variable \(v\) is on the right-hand-side of a statement, \(v\) has a use at this statement. A reaching definition for a given instruction is another instruction, the target variable of which reaches the given instruction without an intervening assignment. We refer to a use \(R_{ij}^v\) (read of variable \(v\) at statement \(i\)) and its reaching definition \(W_{ij}^v\) (write of variable \(v\) at statement \(j\)) as a DefUse pair, denoted as \(W_{ij}^v \rightarrow R_{ij}^v\).

- **Potential DefUse pair**: Let \(W_{ij}^v\) be a write to variable \(v\) at statement \(i\) and \(R_{kj}^v\) a read of at statement \(j\). A static analysis usually cannot decide \(W_{ij}^v \rightarrow R_{kj}^v\) as the analysis is not precise. We define a potential DefUse pair, denoted as \(W_{ij}^v \rightarrow R_{kj}^v\), to indicate the definition at \(W_{ij}^v\) may reach \(R_{kj}^v\) if during an execution Line \(j\) may happen after Line \(i\), and there is no other writes to \(v\) that definitely happen between the two lines. It is an over-approximation so we have \(W_{ij}^v \rightarrow R_{kj}^v\) if \(W_{ij}^v \rightarrow R_{kj}^v\), but not vice versa.

- **Explicit DefUse pair**: A DefUse pair that is encountered during a concrete execution.

- **Implicit DefUse pair**: A DefUse pair that is inferred by the symbolic analysis component.

- **Escort branch pair**: A branch \(b_{ip}^{T/F}\) dominates an instruction \(s\) if an execution of \(b_{ip}^{T/F}\) inevitably leads to the execution of \(s\). The superscript indicates whether the statement at Line \(p\) takes the True or False branch. Let \(B_{ip}^{T/F}\) denote the set of statements dominated by \(b_{ip}^{T/F}\). As shown in Figure 2, \(b_{ip}^{T}\) dominates Lines 4 and 5 and \(B_{ip}^{F}\) includes...
them. Two branches \( b_p^{T/F} \) and \( b_q^{T/F} \) is called an escort branch pair, denoted as \( b_p^{T/F} \Rightarrow b_q^{T/F} \), if there exists \( W_i^p \in B_p^{T/F} \land W_i^q \in B_q^{T/F} \), and \( W_i^p \Rightarrow W_i^q \). Note that there can be more than one pair of shared variable accesses dominated by the two branches.

The formulas are used to infer program behavior under input/thread schedule vectors in our description.

As depicted in Figure 3, our DefUse testing framework, STEM, integrates static analysis, dynamic analysis and symbolic analysis. Given a multithreaded C program, STAM aims to produce a DefUse database that can present the DefUse pairs to users or answer queries from users.

The goal of the static analysis component is to locate escort branch pairs that are later used to steer the dynamic executions. In Section III-A we describe the intuition why they are useful. The procedure terminates once all the escort branch pairs are considered.

Except the first execution that is random, the dynamic analysis component conducts guided executions under an input/thread schedule vector that is computed by symbolic analysis. The DefUse pairs encountered during the concrete execution are recorded. This component passes two pieces of information to the symbolic analysis component: the executed trace \( \pi \) and a set of alternate branches. An alternate branch, defined in Section III-D, is used to compute an input/thread schedule vector that guides a future execution.

Symbolic analysis encodes a trace \( \pi \) as first order logic formulas that can be solved by off-the-shelf SMT solvers [20]. The formulas are used to infer program behavior under input and thread scheduling different from \( \pi \). DefUse pairs that are hidden from dynamic analysis can be discovered by such predicative analysis. An appropriate encoding of \( \pi \) and the alternate branches are used to decide a future execution. The solution to such formula, if satisfiable, represents a pair of input and thread schedule vectors. An execution under the input/schedule vector leads to an execution that is not only different from previous explicitly and implicitly explored paths but also may reveal new DefUse pairs.

In the rest of the section, we illustrate the key steps of our algorithm through a motivating example that is given in Figure 2. There are three threads \( T_0, T_1, T_2 \) with shared variables \( x, y, z \) and local inputs \( a, b, c \). In this paper we represent branches with their line numbers as subscripts and \( T \) or \( F \) as superscript. For example, \( b_p^T \) denotes the else branch of the if-then-else statement at Line 6. If we do not know whether then or else branch is taken at branch \( p \), we represent it as \( b_p^{T/F} \), and use \( b_p^T \) to denote its negation.

In the following we list the symbols that are in the algorithm and the running example. All the sets are initially empty except \( \Omega \).

\[
\begin{align*}
\Omega & = \emptyset \\
\text{for } \Gamma & : \text{the set of test cases that produce actual DefUse pairs} \\
\text{for } \Gamma^x & : \text{the set of explicit DefUse pairs that are detected during the execution of } \pi \\
\text{for } \Gamma^m & : \text{the set of implicit DefUse pairs that are computed based on a symbolic predicative analysis of } \pi \\
\text{for } \Sigma & : \text{the set of escort branch pairs of the program under testing} \\
\text{for } \Sigma_{\pi} & : \text{the set of escort branch pairs covered by } \Gamma^x \text{ of } \pi \\
\text{for } \Sigma_{\pi}^m & : \text{the set of escort branch pairs covered by } \Gamma^m \text{ of } \pi \\
\text{for } \Sigma_{\pi}^A & : \text{the set of escort branch pairs if a branch of } \pi \text{ was reversed} \\
\text{for } W_S & : \text{the working set of to-be-covered escort branch pairs detected by the execution of } \pi.
\end{align*}
\]

A. Static Analysis

The whole program analysis component computes a set \( \Sigma \) of the escort branch pairs. For a \( b_p^{T/F} \Rightarrow b_q^{T/F} \in \Sigma \), the alternate branch computation component in symbolic analysis strives to produce a path \( \pi \) that explores both \( b_p^{T/F} \) and \( b_q^{T/F} \).

For the running example shown in Figure 2, we have \( \Sigma = \{ b_5^T \Rightarrow b_{10}^T, b_5^F \Rightarrow b_{10}^F, b_6^T \Rightarrow b_{10}^F, b_6^F \Rightarrow b_{10}^T, b_8^T \Rightarrow b_{10}^T, b_8^F \Rightarrow b_{10}^F \} \). The reason that \( b_5^T \not\Rightarrow b_5^F \) is because no potential DefUse pairs are dominated by the two branches.

B. Dynamic Analysis

We maintain a set \( \Omega \) to guide dynamic executions. An item \( (\alpha, \beta) \in \Omega \) consists of an input vector \( \alpha \) and a (thread) schedule vector \( \beta \). In our algorithm \( \beta \) is usually partial so an execution under \( (\alpha, \beta) \) has a deterministic prefix and then becomes non-deterministic afterwards. Consider the example
in Figure 2, the input/schedule vector pair \((a = 1, b = 2, c = 3), (T_0L_1, T_0L_4, T_1L_6, T_1L_9, T_1L_{10}, T_1L_{13})\) leads to an execution \(1^F, 4, 6^F, 9, 10^F, 13, 14^F, 17, 5\). The execution is not unique as the thread scheduling becomes random after the first six steps. Another execution \(1^F, 4, 6^F, 9, 10^F, 13, 5, 14^F, 17\) is also valid. However, any execution produced by the input/schedule vector pair guarantees to cover the escort branch pair \(b_f^1 \Rightarrow b_0^1\). Our algorithm ensures consistency between \(\alpha\) and \(\beta\) so there is at least one feasible execution. For example, \((a = 1, b = 2, c = 3), (T_0L_1, T_0L_2)\) is not valid for the running example.

Initially \(\Omega\) contains one item that assigns random values to program input without any restriction on the thread schedule, which leads to a random execution. It’s like performing a concordic execution that treats program variables as symbolic variables along a concrete execution path. More input/schedule vectors are added to \(\Omega\) by symbolic analysis. Figure 4 gives the first random execution trace \(\pi_1\) under the input/schedule vector \((a = 2, b = 2, c = 3), ()\) in SSA (Single Static Assignment) form for the example shown in Figure 2. The explicit DefUse pairs are \(\Gamma^{ex}_{\pi_1} = \{W_0^\phi \Rightarrow R_2^f, W_0^\phi \Rightarrow R_{10}^f, W_2^\phi \Rightarrow R_{13}^f\}\), where \(W_0^\phi\) indicates the initial assignment to \(x\) and \(R_{10}^f\) indicates the use for \(x\) at Line 10. These DefUse pairs are recorded in \(\Gamma\). We project \(\Gamma^{ex}_{\pi_1}\) to the branch pair \(\Sigma^{ex}_{\pi_1} = \{b_f^1 \Rightarrow b_{10}^1\}\), which is removed from \(\Sigma\).

### Algorithm 1: Path GuidedExecution (Program \(P\), Branch-PairSet \(\Sigma\), InputScheduleSet \(\Omega\), DefUseSet \(\Gamma\), TraceSet \(Tested\))

1. \((\hat{\alpha}, \hat{\beta}) = \Omega.remove();\
2. \(\pi = \text{Execute}(P, \hat{\alpha}, \hat{\beta});\
3. \text{if } \pi \in Tested \text{ then }\
4. \quad \text{return NULL;}
5. \text{else}
6. \quad Tested.add(\pi)
7. \quad \text{Let } \Gamma^{\pi} \text{ be the set of DefUse pairs covered in } \pi;
8. \quad \text{Let } \Sigma^{ex} \text{ be the set of branch pairs covered in } \pi;
9. \quad \Sigma^{ex} = \text{BranchPair}(\Gamma^{ex}_{\pi})
10. \quad \Sigma = \Sigma - \Sigma^{ex}
11. \quad \text{return } \pi
12. \text{end if}

### Algorithm 2: PredicativeAnalysis (Trace \(\pi\), DefUsePairSet \(\Gamma\))

1. \(\Gamma^{im} = \Sigma^{im} = \emptyset\);  
2. Encode partial order constraint \(\varphi_{po}\);  
3. Encode program semantics constraint \(\varphi_{sm}\);  
4. Encode interleaving matching constraint \(\varphi_{im}\);  
5. \(\varphi = \varphi_{sm} \land \varphi_{po} \land \varphi_{im}\);  
6. for each \(W_i^x \rightarrow R_j^z \in \Sigma_{\pi}\) do  
7. \quad Encode DefUse constraint \(\varphi_{du}\) for \(W_i^x \rightarrow R_j^z\);  
8. \quad if \(\varphi \land \varphi_{du}\) is satisfiable then  
9. \quad \Gamma^{im} = \Gamma^{im} \cup \{W_i^x \rightarrow R_j^z\};  
10. \end if  
11. end for  
12. \(\Sigma^{im} = \text{EscortBranchPair}(\Gamma^{im})\);  
13. \(\Sigma = \Sigma - \Sigma^{im}\);  
14. \(\Gamma = \Gamma \cup \Gamma^{im}\);

In a multi threaded program, the number of paths under a fixed input can be exponential due to non-deterministic thread schedules. It is infeasible to explicitly execute all such interleavings. In order to alleviate the problem we exploit predicative analysis to enlarge the effectiveness of an execution. The effect of such analysis on \(\pi\) is the detection of the DefUse pairs that are not manifested in \(\pi\). For example, the execution trace \(\pi_1\) in Figure 4 shows that \(W_{13}^x \not\rightarrow R_2^z\), as the write happens after the read. However, predicative analysis on \(\pi_1\) is
able to confirm \( W_{13}^2 \rightarrow R_{2}^2 \). In fact, as shown in Table I, the predicative analysis component detects two implicit DefUse pairs that are not manifested in \( \pi_1 \).

The predicative analysis component transforms an execution trace \( \pi \) into a quantifier free first order logic formula:

\[
\varphi_\pi = \varphi_{po} \land \varphi_{sm} \land \varphi_{im},
\]

where \( \varphi_{po} \), \( \varphi_{sm} \), and \( \varphi_{im} \) denote the partial order constraint, program semantics constraint and interleaving matching constraint, respectively. Let \( \varphi_{du} \) be DefUse constraint that constrains the ordering if a DefUse pair is valid. The intersection \( \varphi_\pi \land \varphi_{du} \) infers program behavior under a different thread scheduling from the actual one executed by \( \pi \), which covers current DefUse pair \( du \).

**Partial Order Constraint (\( \varphi_{po} \)).** This constraint specifies the potential ordering of the instructions in an execution trace \( \pi \). In this paper, we consider sequential consistency memory model only. Let \( o_i \) represent the possible location of instruction \( i \) in a valid execution. If instruction \( i \) happens before instruction \( j \) in \( \pi \) and both belong to the same thread, we enforce \( o_i < o_j \). For example, the partial order constraint for the SSA trace given in Figure 4 is \( \varphi_{po} \):

\[
\begin{align*}
o_1 &< o_2 \land \\
o_6 &< o_9 < o_{10} < o_{13} \land \\
o_{14} &< o_{17}.
\end{align*}
\]

Note that it does not mandate Line 1 happens before Line 6 or Line 14.

Besides the intra-thread ordering, the inter-thread ordering is guarded by synchronization primitives. In multithreaded programs, the most popular synchronization operations are `lock/unlock` and `wait/signal`. Consider two `lock/unlock` pairs on the same mutex. The following constraint mandates that one pair must be executed either before or after another:

\[
\varphi^{LM}_{po} = \bigwedge_{L[m]} o(u_k) < o(l_k) \lor o(u_k) < o(l_i)
\]

where \( L[m] \) denotes the set of `lock/unlock` pairs on mutex \( m \), and \( o(x) \) represents the order of synchronization operation \( x \).

Given a condition variable \( cd \), let \( WT \) be the set of wait operations on \( cd \), and \( SG \) the set of signal operations on \( cd \). The constraint for `wait/signal` is:

\[
\varphi^{W/S}_{po} = \bigwedge_{w \in WT} \bigwedge_{s \in SG} \{ o(w) < o_s < o_w \land m^w_s = 1 \}
\]

Equation 8 gives a formula that enumerates three possible read-write relations for \( x^r \) and its corresponding writes in \( \pi_1 \).

**DefUse Constraint (\( \varphi_{du} \)).** For a read \( r \) and a write \( w \) that do not form a DefUse pair in \( \pi \), they may form a DefUse pair having the input or thread schedule been changed. In order to find such pairs we first scan an executed trace to locate potential DefUse pairs. For example, the set of potential DefUse pairs in \( \pi_1 \) given in Figure 4 is \( \{ W^0_9 \rightarrow R^2_{13}, W^1_9 \rightarrow R^2_{13} \} \).

Let \( \varphi_{du} \) be the set of potential DefUse pairs in \( \pi \). Equa-
tion 9 gives the formula for DefUse constraint. Partial DefUse constraint for \( \pi_1 \) in Figure 2 is given in Equation 10. After solving the two DefUse constraint formulas with Z3 [20] in Equation 10, we get two feasible implicit DefUse pairs from their solutions, which are collected into \( \Gamma^m = \{ R^z_\pi \rightarrow R^z_\pi, W^z_\pi \rightarrow R^z_\pi \} \). Then we project \( \Gamma^m \) to the branch pair \( \Sigma^m_{\pi_1} = \{ b^F \Rightarrow b^T \} \), which is also removed from \( \Sigma \).

\[
\varphi_{du}[W^v \rightarrow R^v] = (o_w < o_r) \land \bigwedge_{x \neq W(W)} (o_x < o_w \lor o_r < o_x) \tag{9}
\]

\[
\varphi_{du}[W^z_\pi \rightarrow R^z_\pi] = o_0 < o_{13} \land (o_{13} < o_2 \lor o_2 < o_0) \tag{10}
\]

### D. Alternate Branch Computation

The goal of the alternate branch computation component is to find to-be-explored executions and compute input/schedule vector pairs that enforce these executions. A to-be-explored path must satisfy the following three properties:

- it has not been executed by dynamic execution, and
- it has not been inferred by predicative analysis, and
- its execution may find new DefUse pairs.

Let \( \pi_B = \{ b_{10}^{F/T} \ldots b_k^{F/T} \} \) be the set of branches in an explored path \( \pi \). For any \( \pi' \) that is inferred from \( \pi \) by predicate analysis, we have \( \pi_B = \pi_B \), even though the orders of the branches in \( \pi \) and \( \pi' \) are different. In addition, for any \( \pi' \) such that \( \pi_B = \pi_B \), \( \pi' \) has the same set of instructions as \( \pi \). According to our algorithm, its implicit DefUse pairs must be inferred by the predicative analysis component. Based on this observation, a path whose set of branches is different from all dynamically executed paths satisfies the first two properties.

Let \( \pi = (\ldots b_{10}^{T/F} \ldots b_k^{T/F} \ldots b_{10}^{T/F} \ldots) \) be an executed trace with \( k \) branches. By reversing \( b_{10}^{T/F} \) to \( b_{10}^{F/T} \), we observe the set of statements \( B^T_{p/F} \) dominated by \( b_{10}^{T/F} \). If there are any potential DefUse pairs between the statements in \( B^T_{p/F} \) and \( B^T_{q/F} \), \( b_{q}^{F/T} \) in \( \pi \) and \( b_{10}^{F/T} \) forms an escort branch pair. Any execution following \( \pi' = (\ldots b_{10}^{F/T} \ldots) \) satisfies the third aforementioned property. By consider one branch in \( \pi \) at a time, we can obtain a set of \( \Sigma^m_{\pi} \) of such escort branch pairs derived from \( \pi \). Consider \( \pi_1 = (1T, 2, 6F, 9, 10F, 13, 14F, 17) \) in the running example, reversing \( T \) or \( F \) leads to potential DefUse pairs with existing branches in \( \pi_1 \). As a result, we have \( \Sigma^m_{\pi_1} = \{ b_1^T \Rightarrow b_{10}^T, b_{10}^T \Rightarrow b_1^F, b_1^F \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^F \} \).

After the dynamic execution of \( \pi \), \( WS_\pi \) tells what to explore but not how to explore. We exploit symbolic computation to compute the input/schedule vectors that lead to executions that cover the escort branch pairs in \( WS_\pi \). Let \( \varphi_{sm} \), \( \varphi_{po} \), and \( \varphi_{im} \) be the semantics constraint, partial-order constraint, and interleaving matching constraint obtained from predicative analysis on \( \pi \). We need to remove the branch terms from \( \varphi_{sm} \) because we need to change the constraint on branch. Besides, we need to remove input values from \( \varphi_{sm} \) in order to computing new inputs. Let \( \varphi_{sm}' \) be the revised semantics constraint without branch terms and input values. Given \( b_{10}^{F/T} \Rightarrow b_{10}^{F/T} \in WS_\pi \), without loss of generality, we assume it is \( b_{10}^{F/T} \) that has been negated in \( \pi \). The encoding that enforces a path that takes \( b_{10}^{F/T} \) is as the following:

\[
\varphi_{bp} = \varphi_{sm}' \land \varphi_{po} \land \varphi_{im} \land b_{10}^{F/T} \land \bigwedge_{1 \leq q \leq k} (o_q < o_p \rightarrow b_q^{F/T}), \tag{11}
\]

where \( o_q \) represents the order of \( b_q \) and the arrow indicates implication. If \( \varphi_{bp} \) is satisfiable, the formula ensures that \( b_q \) is the first negated branch in \( \pi \) and all the branches before \( b_q \) remains the same. Note that if there exists any branch \( b_q \) that is negated before \( b_{10} \), it is not guaranteed that \( b_q \) can be negated or even visited. The solution to Equation 11 contains the assignments to program input \( \vec{a} \) and thread schedule \( \vec{\beta} \). An execution under the pair of vectors \( (\vec{a}, \vec{\beta}) \), leads to an execution that executes \( b_{10}^{F/T} \).

Back to the example in Figure 2, \( WS_{\pi_1} = \Sigma^m_{\pi_1} \cap \Sigma = \{ b_1^T \Rightarrow b_{10}^T, b_1^F \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^F \} \). By projecting \( WS_{\pi_1} \) to branches, we can confirm there are two to-be-negated branches \( b_1^T \) and \( b_{10}^F \). Then we build their negation constraint formulas and verify them one by one. For the negation \( b_1^T \) to \( b_{10}^F \) as shown in Equation 12, \( \varphi_{bp}[b_{10}^F \rightarrow b_1^T] \) is unsatisfiable since \( b_{10} \) can’t be negated in \( \pi_1 \) at all. Therefore, we can’t compute a feasible input/schedule vector pair. For the negation \( b_{10}^F \rightarrow b_1^T \) (omitting its equation), one input/schedule vector pair \((1, 2, 3, \{1F\})\) is computed from its satisfiable solution and leads to the new execution that covers the branch pairs \( b_1^F \Rightarrow b_{10}^F \) and \( b_{10}^F \Rightarrow b_1^F \).

\[
\varphi_{bp}[b_{10}^F \rightarrow b_1^T] = x_2^2 > 2 \land (o_1 < o_{10} \rightarrow a_0 > 1) \tag{12}
\]

### E. Overall Algorithm

Algorithm 4 gives the overall procedure that integrates the four key components.

- **ProgramBranchPairs** performs static analysis on the program \( P \) and returns the set of branch pairs \( \Sigma \).
Algorithm 4 DefUseTesting (Program P)
1: \( \Gamma = \text{Tested} = \emptyset \);
2: \( \Sigma = \text{ProgramBranchPairs}(P) \);
3: \( \Omega = \{ \{ \text{random value}, \emptyset \} \} \);
4: while \( \Sigma \neq \emptyset \land \Omega \neq \emptyset \) do
5: \( \pi = \text{GuidedExecution} (P, \Sigma, \Omega, \Gamma, \text{Tested}) \);
6: if \( \pi = \text{NULL} \) then
7: \( \text{Continue} \);
8: else
9: \( \text{PredicativeAnalysis} (\pi, \Gamma) \);
10: \( \text{AlternateBranchComputation} (P, \pi, \Sigma, \Omega) \);
11: end if
12: end while
13: return \( \Gamma \);

- \text{GuideExecution} executes \( P \) under an input/schedule vector pair \((\vec{\alpha}, \vec{\beta})\) removed from \( \Omega \). The branch pairs covered by the executed trace \( \pi \) are removed from \( \Sigma \), and the DefUse pairs discovered in \( \pi \) are added to \( \Gamma \).
- \text{PredicativeAnalysis} conducts symbolic analysis on a given trace \( \pi \). The computed DefUse pairs are added to \( \Gamma \).
- \text{AlternateBranchComputation} computes branch pairs in \( P \) by negating one branch at a time in \( \pi \). After eliminating the branch pairs that do not need consideration and that have not been considered before, we compute the input/schedule vector pairs that include both branch pairs in a path.

Table I gives the complete procedure for the computation of DefUse pairs in the 3-threaded program shown in Figure 2.

- Step 0: Static analysis produces the set of escort branch pairs \( \Sigma = \{ b_1^F \Rightarrow b_1^T, b_1^T \Rightarrow b_1^{T0}, b_1^F \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F, b_1^T \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F \} \).
- Step 1: A random input and thread schedule \(((2, 2, 3), \emptyset)\) leads to the execution of \( \pi_1 \), which detects the four DefUse pairs in \( \Gamma_{\pi_1}^{ex} \). A symbolic predicative analysis confirms that four other DefUse pairs in \( \Gamma_{\pi_1}^{im} \) exist in \( \pi_1 \) once its input and thread schedule are changed. The explicitly detected and implicitly computed DefUse pairs, eight in total, are recorded in \( \Gamma \). The branch pairs covered by \( \Gamma_{\pi_1}^{ex} \) and \( \Gamma_{\pi_1}^{im} \) are \( \Sigma_{\pi_1}^{ex} = \{ b_1^T \Rightarrow b_1^{T0} \} \) and \( \Sigma_{\pi_1}^{im} = \{ b_1^{T0} \Rightarrow b_1^T \} \), respectively. We remove the items in \( \Sigma_{\pi_1}^{ex} \) and \( \Sigma_{\pi_1}^{im} \) from \( \Sigma \), which gets \( \Sigma = \{ b_1^F \Rightarrow b_1^{T0}, b_1^T \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F \} \). From \( \pi_1 \), we may obtain a different execution by reverting either \( b_1^F \) or \( b_1^{T0} \), which means \( \Sigma_{\pi_1}^{A} = \{ b_1^F \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F, b_1^T \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F \} \).

- Step 2: The removed item \(((1, 2, 3), (1F))\) from \( \Omega \) causes the execution of \( \pi_2 \), which leads to the detection of 3 explicit DefUse pairs in \( \Gamma_{\pi_2}^{ex} \) and 0 implicit DefUse pairs \( \Gamma_{\pi_2}^{im} \). The new detected items are added to \( \Gamma \). Since \( \pi_2 \) covers \( b_1^T \Rightarrow b_1^{T0} \), such item is removed from \( \Sigma \). If we revert the branches in \( \pi_2 \), we may cover \( \Sigma_{\pi_2}^{A} = \{ b_1^T \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F, b_1^F \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F \} \).

- Step 3: By following a removed input/schedule pair vector \(((1, 2, 3), (1F, 4, 6F, 9, 10F))\) from \( \Omega \) we obtain \( \pi_3 \). The dynamic execution and predicative analysis detect 4 new DefUse pairs that are added to \( \Gamma \). The branch pair covered by \( \pi_3 \) is \( \{ b_1^F \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F \} \), which is removed from \( \Sigma \). By reverting a branch in \( \pi_3 \), we may cover \( \{ b_1^T \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F, b_1^F \Rightarrow b_1^{T0}, b_1^{T0} \Rightarrow b_1^F \} \). However, both have been covered before \( WS_{\pi_3} = \emptyset \). Thus \( \pi_3 \) does not produce any to-be-covered paths.

- Step 4: The last execution \( \pi_4 = ((1T, 2, 6F, 9, 10F, 13, 14F, 17) \) is determined by the remaining one item \(((3, 2, 3), (1T)) \) in \( \Omega \). It equals \( \pi_1 \) and has been already analyzed or tested. So we ignore it and skip. However, since \( \Omega \) are empty now, the algorithm terminates. We consider the items in \( \Sigma \) as invalid branch pairs, which can’t be covered in our exploration.

IV. EXPERIMENTS

We have implemented the proposed method in a software tool STEM that is built upon LLVM [21], KLEE [5] and Z3 [20]. It targets multithreaded C programs implemented with the POSIX thread library. Before exploring DefUse pairs, we first recognize shared access points with alias analysis. Then we add a scheduler, a listener and an encoder into KLEE. The scheduler is able to guide program executions under a given thread interleaving specification. After collecting executed instruction instances by the listener, the encoder translates a trace into a logic formula. In total, we have added 7,500 lines of code to KLEE.

Our empirical study is conducted on eleven benchmarks that are obtained from well-known application suites SPLASH2 [22] (fft, radix, lu_contiguous and lu_non_contiguous) and PARSEC [23] (blackscholes), as well as experimental objects in previous studies on bounded model checker ESBMC [24] (including account, banking, twostage, lazy, micro, reorder, stateful, token, arithmetic, queue, and stack) and a trace simplification technique [25] (pfscan). In total we have 22 buggy versions, each of which is inserted one assertion failure. Some programs have two buggy versions, such as fft and lu-c. These bugs are only triggered by special thread scheduling and barely arise in most executions. We set part of their inputs or shared variables as symbolic variables, which may affect branch choosing. But not all programs in SPLASH2 and PARSEC are suitable for our prototype tool, because of non-linear computation or extremely long execution traces.

Table II gives the experimental results of aforementioned benchmarks by STEM in detail. In the table, Column LOC gives the line of source code and Column #Inst shows the
## Table I

The computation steps for the multithreaded program shown in Figure 2, where GE stands for Guided Execution, and PA Predicative Analysis, ABC Alternate Branch Computation. Items marked as red stand for $\Delta$ of DefUse pairs.

<table>
<thead>
<tr>
<th>Step</th>
<th>Component</th>
<th>Set Content</th>
</tr>
</thead>
</table>
| 1 | Guided Execution | $I : ((2, 3), (\langle \rangle)) \rightarrow \pi_1 : \{1T, 2, 6F, 9, 10F, 13, 14F, 17\}$ (New)  
$\Gamma_{\pi_1}^{ex} = \{W_0^0 \rightarrow R_2^x, W_0^2 \rightarrow R_{10}^x, W_2 \rightarrow R_{13}^x\}$  
$\Sigma_{\pi_1}^{ex} = \{b_1^F \Rightarrow b_{10}^0\}$  
$\Sigma = \Sigma - \Sigma_{\pi_1}^{ex} = \{b_1^F \Rightarrow b_{10}^0, b_1^F \Rightarrow b_{10}^0, b_1^F \Rightarrow b_{10}^0, b_{10}^0 \Rightarrow b_1^F\}$  
$\Delta \Gamma = \{W_0^0 \rightarrow R_2^x, W_0^2 \rightarrow R_{10}^x, W_2 \rightarrow R_{13}^x\}$  
$WS_{\pi_1} = \Sigma_{\pi_1}^{A} \cap \Sigma = \{b_1^F \Rightarrow b_{10}^0, b_1^F \Rightarrow b_{10}^0, b_{10}^0 \Rightarrow b_1^F\}$  
$\Omega = \{((1, 2, 3), (1F))\}$ |
| | Predicative Analysis |  |
| | Alternate Branch Computation |  |
| 2 | Guided Execution | $I : ((1, 2, 3), (1F)) \rightarrow \pi_2 : \{1F, 4, 5, 6F, 9, 10T, 11, 14F, 17\}$ (New)  
$\Gamma_{\pi_2}^{ex} = \{W_0^0 \rightarrow R_2^x, W_2 \rightarrow R_{11}^x, W_5^x \rightarrow R_{10}^x\}$  
$\Sigma_{\pi_2}^{ex} = \{b_1^F \Rightarrow b_{10}^0\}$  
$\Sigma = \Sigma - \Sigma_{\pi_2}^{ex} = \{b_1^F \Rightarrow b_{10}^0, b_1^F \Rightarrow b_{10}^0, b_1^F \Rightarrow b_{10}^0\}$  
$\Delta \Gamma = \{W_0^0 \rightarrow R_2^x, W_2 \rightarrow R_{11}^x, W_5^x \rightarrow R_{10}^x\}$  
$WS_{\pi_2} = \Sigma_{\pi_2}^{A} \cap \Sigma = \{b_1^F \Rightarrow b_{10}^0, b_1^F \Rightarrow b_{10}^0, b_{10}^0 \Rightarrow b_1^F\}$  
$\Omega = \{((1, 2, 3), (1F)), ((3, 2, 3), (1T))\}$ |
| | Predicative Analysis |  |
| | Alternate Branch Computation |  |
| 3 | Guided Execution | $I : ((1, 2, 3), (1F, 4, 6F, 9, 10F)) \rightarrow \pi_3 : \{1F, 4, 6F, 9, 10F, 5, 13, 14F, 17\}$ (New)  
$\Gamma_{\pi_3}^{ex} = \{W_0^0 \rightarrow R_2^x, W_0^2 \rightarrow R_{13}^x, W_4^x \rightarrow R_{10}^x\}$  
$\Sigma_{\pi_3}^{ex} = \{b_1^F \Rightarrow b_{10}^0\}$  
$\Sigma = \Sigma - \Sigma_{\pi_3}^{ex} = \{b_1^F \Rightarrow b_{10}^0\}$  
$\Delta \Gamma = \{W_0^0 \rightarrow R_2^x, W_0^2 \rightarrow R_{13}^x, W_4^x \rightarrow R_{10}^x\}$  
$WS_{\pi_3} = \Sigma_{\pi_3}^{A} \cap \Sigma = \{b_1^F \Rightarrow b_{10}^0\}$  
$\Omega = \{((3, 2, 3), (1T))\}$ |
| | Predicative Analysis |  |
| | Alternate Branch Computation |  |
| 4 | Guided Execution | $I : ((3, 2, 3), (1T)) \rightarrow \pi_4 : \{1T, 2, 6F, 9, 10F, 13, 14F, 17\}$ (Tested)  
$\Sigma = \{b_1^F \Rightarrow b_{10}^0\}$  
$\Omega = \emptyset$ (Stopped) |
| | Alternate Branch Computation |  |
average number of instructions in an execution trace. Column \#F and \#Tr list the number of encoding formulas and the number of trace during exploration, respectively. Column \#BP shows the number of covered branch pairs. Column \#DU, \#imDU and \#inDU give the total number of explored DefUse pairs, the number of implicit DefUse pairs and the number of invalid potential DefUse pairs, respectively. Column Time and Mem present the time usage and memory consumption, respectively. In particular, Column solTime lists the SMT solving time. Column BT gives the result of bug finding, where ○, • and × stand for the bugs triggered during guided execution, the bugs detected during replaying the implicit DefUse pairs, and the bugs that are missed, respectively.

As shown in Table II, on average about two traces are explored under the guidance of alternate branch pairs. From Column \#imDU, we observe that implicit DefUse pairs just account for a small percentage, about 10%, of the total DefUse pairs. Note that we only consider the relatively large programs, which make programs more error-prone. Our experimental data indicate that our benchmarks are well-written. On the other hand, hard-to-find bugs are hidden in rarely executed interleavings. Thus it is necessary to find the implicit DefUse pairs.

Table II also shows that most time is spent on SMT solving. This points out a direction of optimizing our method, such as simplifying formulas. A potential DefUse pair is an implicit one if it is satisfiable in symbolic analysis. Otherwise, it is an invalid one that is included in Column \#inDU. On average the number of invalid DefUse pairs accounts for 59%(\#inDU/(\#imDU+\#inDU)) of the total potential DU pairs. Therefore, we can greatly optimize our method if we can recognize infeasible potential DefUse pairs before symbolic analysis. Except blackscholes, all of the bugs are detected during the concrete execution. For programs queue_bug, stack_bug and token_ring_bug, their bugs are triggered during executing an input/schedule vector that explores a new branch. And the remaining 18 bugs are triggered during replaying the explored implicit DefUse testcases.

V. Threats to Validity

There are still some challenges in our current work. For example, the static pointer analysis may give imprecise result of escort branch pairs. Without the runtime information, some feasible branch pairs can be missed. Meanwhile, it is possible that an invalid branch pair is considered but can never be covered during the computation. In addition, SMT solvers has limited capability in handling nonlinear expressions and large formulas. To make this approach scalable we have to design aggressive optimization heuristics for constraint simplification. Despite implementation challenges, we believe DefUse coverage is a better coverage criteria than path/interleaving coverage.

VI. Related Work

In this section, we discuss the most relevant work to ours, including DefUse, symbolic execution, concurrency testing and symbolic analysis.
**Definition-Use.** DefUse relations have been used for software testing. Many works [26, 27] have focused on testing DefUse relation to increase test coverage. Y. Shi et al. developed a tool to test and detect bugs in both concurrent and sequential programs by using analyzing DefUse invariants [11]. Their tool automatically extracts the invariants from programs that are executed many times under some fixed inputs. Su et al. implemented CAUT to generate inputs for statically extracted definition-uses [28]. CAUT explores the path covering current DefUse with symbolic execution and recognizes the infeasible DefUse using model checking. However, it is only suitable for sequential programs.

**Symbolic Execution.** In recent years symbolic execution has become an important technique for effectively testing programs [1–6, 9, 29–33]. For example, KLEE, an popular symbolic execution tool, is capable of automatically generating testcases on complex and environmentally-intensive programs and achieving high coverage [5]. V. Chipounov et al. presented a platform called $S^2E$, which can test programs at binary level and scale to large systems using selective symbolic execution and execution consistency model [29]. S. Bucur et al. implemented the first cluster-based parallel symbolic execution platform, cloud9, which symbolically executes program on distributed nodes [30]. Cloud9 supports multithreaded programs using POSIX model and provides standard code both coverage interleaving guarantees. However, it enumerates the thread interleavings that only consider context switch on synchronization statements.

Meanwhile, there is a large body of work on mitigating path explosion in symbolic execution, including the use of function summaries [34], may-must abstraction [35], demand-driven refinement [36], state matching [37], state merging [38], structural coverage [39], weakest precondition computation [33], and dependence guide [40]. McMillan proposed ac method called lazy abstraction with interpolants [41, 42], which has been shown to be effective in model checking sequential software. Jaffar et al. [43] used a similar method in the context of constraint programming to compute resource-constrained shortest paths and worst-case execution time. However, a direct extension of such methods to multithreaded programs would be inefficient since they lead to the naive exploration of all thread interleavings. Some recent work [9, 31, 32] focused on the symbolic execution of multithreaded programs, but they are not scalable due to path/interleaving explosion.

**Concurrency Testing.** For testing concurrent bugs, the most direct and scalable method is random testing which inserts random delays into execution at certain memory access point in order to exercise different interleavings [44, 45]. However, due to the random nature of these techniques, they are not able to reveal concurrency bugs occurring under specific interleavings [46]. To explore a unique scheduling in each run, Recent work [19, 47, 48] systematically test concurrent programs via steering thread scheduler according to context switch bound or shared access point relation. Much more work focus on the detection of specific bugs, such as data race [13–15], atomicity [16, 46, 49], and dead lock [50, 51]. The common shortage in above work is that only small part of interleavings are investigated under a fixed input.

**Symbolic Analysis.** Because of recent significant advances in SMT solver, more and more people begin to reduce various analysis problems about multithreaded programs to constraint solving. A. Farzan et al. present Exception Rifle, a constraint-solving based tool, which finds schedule-sensitive branches from real concurrent systems by deriving constraints for each to-be-negated branch and solving these with an SMT solver [54]. However, the approach mealy finds alternate branches under a current trace. Our approach, instead, explores alternate branches to analyze newly discovered traces.

**VII. Conclusion**

In this paper we presented a framework that conducts systematic testing of DefUse data flow for multithreaded programs. We have implemented a tool called STEM and the initial evaluation show the benefit of data flow testing. Compared with path/interleaving testing, the DefUse testing not only is more scalable but also leads to better comprehension of concurrent programs. For future work, we plan to design a query language so users can query data-flow related properties. This also requires appropriate presentation format as a deterministic replay is needed to show a trace include a particular data-flow pattern.

**VIII. Acknowledgements**

This work was supported by National Key Research and Development Program of China (2016YFB0800202), the National Natural Science Foundation of China (91218301, 11301254, 91418205, 61472318, 61428206, 61532015, 61632015), Fok Ying-Tong Education Foundation (151067), Key Project of the National Research Program of China (2013BAK09B01), Ministry of Education Innovation Research Team (IRT13035), the Fundamental Research Funds for the Central Universities and the National Science Foundation (NSF) under grant DGE-1522883.

**References**


[46] S. Park, S. Lu, and Y. Zhou, “CTrigger: Exposing Atomicity...


