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 scrutinization. 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

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.23

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.

T_0	T_1	T_2
$void*\ t1(a)\{$	$void*\ t2(b)\{$	$void*\ t3(c)\{$
$if(a>1)$	$if(b>1)$	$if(c>1)$
$\quad x=x+2;$	$\quad x=x*2;$	$\quad x=x^2;$
$else$	$else$	$else$
$\quad x=x+3;$	$\quad x=x*3;$	$\quad x=x^3;$
$if(x>8)$	$if(x>8)$	$if(x>8)$
$\quad x=x*2;$	$\quad x=x^2;$	$\quad x=x+2;$
$else$	$else$	$else$
$\quad x=x*3;\}$	$\quad x=x^3;\}$	$\quad x=x+3;\}$

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.

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 (4×4), and the total number of paths to enumerate all the DefUse is 48 (16×3). 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_i^v (read of variable v at statement i) and its reaching definition W_j^v (write of variable v at statement j) as a `DefUse pair`, denoted as $W_j^v \rightarrow R_i^v$.

`Potential DefUse pair`: Let W_i^v be a write to variable v at statement i and R_j^v a read of at statement j. A static analysis usually cannot decide $W_i^v \rightarrow R_j^v$ as the analysis is not precise. We define a `potential DefUse pair`, denoted as $W_i^v \rightharpoonup R_j^v$, to indicate the definition at W_i^v *may* reach R_i^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 $W_i^v \rightharpoonup R_j^v$ if $W_i^v \rightarrow R_j^v$, 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 $b_p^{T/F}$ *dominates* an instruction s if an execution of $b_p^{T/F}$ inevitably leads to the execution of s. The superscript indicates whether the statement at Line p takes the `True` or `False` branch. Let $B_p^{T/F}$ denote the set of statements dominated by $b_p^{T/F}$. As shown in Figure 2, b_1^F dominates Lines 4 and 5 and B_1^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 $W_i^v \in B_p^{T/F} \wedge R_j^v \in B_q^{T/F}$, and $W_i^v \rightharpoonup R_j^v$. Note that there can be more than one pair of shared variable accesses dominated by the two branches.

```
   T0                      T1                      T2
void * t0(a){          void * t1(b){          void * t2(c){
1 :  if(a > 1)//b1     6 :  if(b > 5)//b6     14 :  if(c < 2)//b14
2 :     z = x + 2;     7 :     b = y + 1;     15 :     d = 1;
3 :  else              8 :  else              16 :  else
4 :     z = x − 2;     9 :     b = y − 1;     17 :     d = 0; }
5 :     x = 3; }       10 :  if(x > 2)#b10
                       11 :     y = z + 1;
                       12 :  else
                       13 :     x = z − 1; }
```

Fig. 2. A multithreaded program with three threads. There are three local inputs a, b, c and three shared variables x, y, z. The initial values are $x = y = z = 0$.

III. DefUse Testing Algorithms

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 π 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 π 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 π. DefUse pairs that are hidden from dynamic analysis can be discovered by such predicative analysis. An appropriate encoding of π 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_6^F 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 $b_p^{F/T}$ 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 Ω.

- Γ: the set of test cases that produce actual DefUse pairs. Its items are in the format of $W_i^x \rightarrow R_j^x @ (\vec{i}, \vec{s})$. For ease of understanding, we omit the input and thread schedule vectors in our description.
- Γ_π^{ex}: the set of explicit DefUse pairs that are detected during the execution of π.
- Γ_π^{im}: the set of implicit DefUse pairs that are computed based on a symbolic predicative analysis of π.
- Ω: the set of input and thread schedule pairs $(\vec{\alpha}, \vec{\beta})$. Produced by the alternate branch computation component, its items guide dynamic executions.
- Σ: the set of escort branch pairs of the program under testing. An item $b_p^{T/F} \Rightarrow b_q^{T/F} \in \Sigma$ indicates potential DefUse pairs in the two branches, which needs confirmation or falsification from dynamic or symbolic analysis.
- Σ_π^{ex}: the set of escort branch pairs covered by Γ_π^{ex} on π.
- Σ_π^{im}: the set of escort branch pairs covered by Γ_π^{im} on π.
- Σ_π^{A}: the set of escort branch pairs if a branch of π was reversed.
- WS_π: the working set of to-be-covered escort branch pairs detected by the execution of π.

A. Static Analysis

The whole program analysis component computes a set Σ 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 π that explores both $b_p^{T/F}$ and $b_q^{T/F}$. If the path is found, W_i^v and R_j^v that are dominated by the pair of branches are both executed in π. Note that we do not require a path that the definition of W_i^v reaches R_j^v. This is because the predicative analysis component can infer program behavior by considering different ordering of the instructions in π. Since in our algorithm dynamic analysis explores new paths with different branches and symbolic analysis explores new paths with different instruction ordering, we only need to record branch information in Σ as the items in the set are used to guide dynamic execution.

For the running example shown in Figure 2, we have $\Sigma = \{b_1^F \Rightarrow b_{10}^T, b_1^T \Rightarrow b_{10}^T, b_1^F \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^F, b_1^T \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^T\}$. The reason that $b_1^T \not\Rightarrow b_6^T$ is because no potential DefUse pairs are dominated by the two branches.

B. Dynamic Analysis

We maintain a set Ω to guide dynamic executions. An item $(\vec{\alpha}, \vec{\beta}) \in \Omega$ consists of an input vector $\vec{\alpha}$ and a (thread) schedule vector $\vec{\beta}$. In our algorithm $\vec{\beta}$ is usually partial so an execution under $(\vec{\alpha}, \vec{\beta})$ has a deterministic prefix and then becomes non-deterministic afterwards. Consider the example

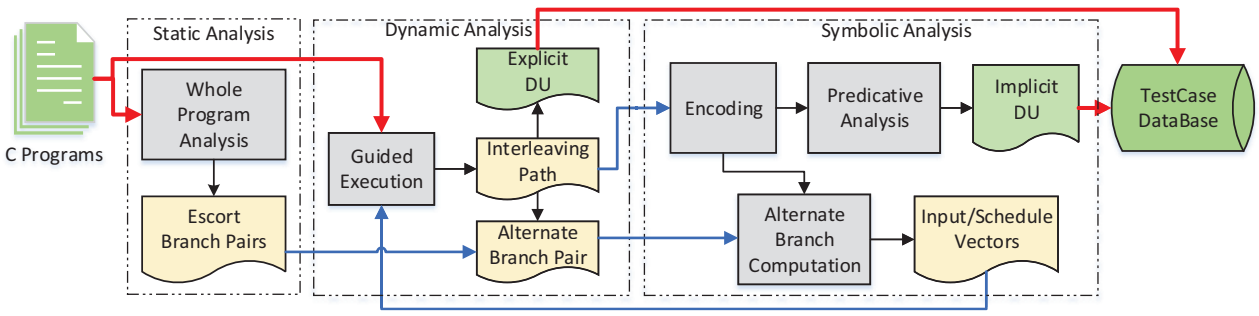

Fig. 3. The Synergistic Analysis Framework for DefUse Testing.

Algorithm 1 Path GuidedExecution (Program P, BranchPairSet Σ, InputScheduleSet Ω, DefUseSet Γ, TraceSet $Tested$)

1: $(\vec{\alpha}, \vec{\beta}) = \Omega$.remove();
2: π = Execute($P, \vec{\alpha}, \vec{\beta}$);
3: **if** $\pi \in Tested$ **then**
4: **return** NULL;
5: **else**
6: $Tested$.add(π) ;
7: Let Γ_π^{ex} be the set of DefUse pairs covered in π;
8: $\Gamma = \Gamma \cup \Gamma_\pi^{ex}$;
9: Let Σ_π^{ex} be the set of branch pairs covered in π;
10: $\Sigma_\pi^{ex} = BranchPair(\Gamma_\pi^{ex})$;
11: $\Sigma = \Sigma - \Sigma_\pi^{ex}$;
12: **return** π;
13: **end if**

in Figure 2, the input/schedule vector pair $(\langle a = 1, b = 2, c = 3\rangle, \langle T_0L_1, T_0L_4, T_1L_6, T_1L_9, T_1L_{10}, T_1L_{13}\rangle)$ leads to an execution $\langle 1^F, 4, 6^F, 9, 10^F, 13, 14^F, 17, 5\rangle$. The execution is not unique as the thread scheduling becomes random after the first six steps. Another execution $\langle 1^F, 4, 6^F, 9, 10^F, 13, 5, 14^F, 17\rangle$ is also valid. However, any execution produced by the input/schedule vector pair guarantees to cover the escort branch pair $b_1^F \Rightarrow b_{10}^F$. Our algorithm ensures consistency between $\vec{\alpha}$ and $\vec{\beta}$ so there is at least one feasible execution. For example, $(\langle a = 1, b = 2, c = 3\rangle, \langle T_0L_1, T_0L_2\rangle)$ is not valid for the running example.

Initially Ω 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 concolic execution that treats program variables as symbolic variables along a concrete execution path. More input/schedule vectors are added to Ω by symbolic analysis. Figure 4 gives the first random execution trace π_1 under the input/schedule vector $(\langle a = 2, b = 2, c = 3\rangle, \langle\rangle)$ in SSA (Single Static Assignment) form for the example shown in Figure 2. The explicit DefUse pairs are $\Gamma_{\pi_1}^{ex} = \{W_0^x \rightarrow R_2^x, W_0^x \rightarrow R_{10}^x, W_2^z \rightarrow R_{13}^z\}$, where W_0^x indicates the intial assignment to x and R_{10}^x indicates the use for x at Line 10. These DefUse pairs are recorded in Γ. We project $\Gamma_{\pi_1}^{ex}$ to the branch pair $\Sigma_{\pi_1}^{ex} = \{b_1^T \Rightarrow b_{10}^F\}$, which is removed from Σ.

$$
\begin{array}{ll}
0: & x_0^w = 0, y_0^w = 0, z_0^w = 0 \\
1: & if(a_0 > 1)//true \\
2: & z_1^w = x_1^r + 2; \\
6: & if(b_0 > 5)//false \\
9: & b_1 = y_1^r - 1; \\
10: & if(x_2^r > 2)//false \\
13: & x_3^w = z_2^r - 1; \\
14: & if(c_0 < 2)//false \\
17: & d_0 = 0;
\end{array}
$$

Fig. 4. SSA form of π_1

C. Predicative Analysis

Algorithm 2 PredicativeAnalysis (Trace π, DefUsePairSet Γ)

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

In a multithreaded 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 π is the detection of the DefUse pairs that are not manifested in π. For example, the execution trace π_1 in Figure 4 shows that $W_{13}^x \not\rightarrow R_2^x$, as the write happens after the read. However, predicative analysis on π_1 is

able to confirm $W_{13}^x \rightarrow R_2^x$. In fact, as shown in Table I, the predicative analysis component detects two implicit DefUse pairs that are not manifested in π_1.

The predicative analysis component transforms an execution trace π into a quantifier free first order logic formula:

$$\varphi_\pi = \varphi_{po} \wedge \varphi_{sm} \wedge \varphi_{im}, \tag{1}$$

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

Partial Order Constraint ***(φ_{po})***. This constraint specifies the potential ordering of the instructions in an execution trace π. 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 π 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 φ_{po} :

$$\begin{array}{l} o_1 < o_2 \wedge \\ o_6 < o_9 < o_{10} < o_{13} \wedge \\ o_{14} < o_{17}. \end{array} \tag{2}$$

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:

$$\varphi_{po}^{L[m]} = \bigwedge_{l_i/u_i, l_k/u_k \in L[m]} o(u_i) < o(l_k) \vee o(u_k) < o(l_i) \tag{3}$$

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:

$$\begin{array}{l} \varphi_{po}^{W[cd]} = \{ \bigwedge_{w \in WT} \bigvee_{s \in SG} (o_w < o_s < o_{w'} \wedge m_s^w = 1)\} \\ \bigwedge \varphi_{SG}^{WT} \bigwedge \varphi_{WT}^{SG} \end{array} \tag{4}$$

where $o_{w'}$ denotes the next event of wait on cd immediately after w in the same thread, $o_w < o_s < o_{w'}$ indicates that a signal operation s must be executed between w and w', and $m_s^w = 1$ flags that s is mapped to w. Equation 5 defines φ_{SG}^{WT} and φ_{WT}^{SG}, in which φ_{SG}^{WT} enforces that each wait operation w needs to map to at least one signal operation, and φ_{WT}^{SG} restricts that each signal operation s signals at most one wait operation.

$$\begin{array}{l} \varphi_{SG}^{WT} = \bigwedge_{w \in WT} \{\{ \sum_{s \in SG} m_s^w\} \geq 1\} \\ \varphi_{WT}^{SG} = \bigwedge_{s \in SG} \{\{ \sum_{w \in WT} m_s^w\} \leq 1\} \end{array} \tag{5}$$

Constraints on other types of synchronization primitives are modeled similarly. The conjunction of these intra- and inter-thread constraints relaxes the total order observed in an execution trace π.

Program Semantics Constraint ***(φ_{sm})***. The constraint maps executed individual instructions to corresponding formula. We skip detailed presentation as it requires mapping rules for complete LLVM syntax. However, the mapping rules are straightforward. Equation 6 gives the program semantics constraint for π_0 shown in Figure 4. Note that φ_{sm} enforces the same control flow as a derivation leads to an execution with instructions unknown to a trace. That is, all the executions inferred from π take the same branches as π.

$$\begin{array}{l} a_0 = 1 \wedge b_0 = 2 \wedge c_0 = 3 \wedge \\ x_0^w = 0 \wedge y_0^w = 0 \wedge z_0^w = 0 \wedge \\ a_0 > 1 \wedge z_1^w = x_1^r + 2 \wedge \\ \neg(b_0 > 5) \wedge b_1 = y_1^r - 1 \wedge \neg(x_2^r > 2) \wedge x_3^w = z_2^r - 1 \wedge \\ \neg(c_0 < 2) \wedge d_0 = 0 \end{array} \tag{6}$$

Interleaving Matching Constraint ***(φ_{im})***. The program semantics constraint considers each thread individually. In multithreaded programs different threads communicate data via shared variables. The complexity of multithreaded programs is due to the non-deterministic nature of such communication. The purpose of interleaving matching constraint is to enumerate all possible matchings between read and write instructions of shared variables. Consider a shared variable v. Let $R(v)$ and $W(v)$ be the sets of reads and writes on v, respectively. We use v_r to denote the read of v at instruction r, and v_w the write of v at instruction w. In addition, let o_r and o_w be the order variables of r and w. Equation 7 gives interleaving matching constraint. It specifies that for a shared variable v, v_r reads the value of v_w if r is executed after w, and there are no other writes to v in between.

$$\begin{array}{l} \varphi_{im} = \bigwedge_{v \in V} \bigwedge_{r \in R(v)} \bigvee_{w \in W(v)} \{(v_r = v_w \wedge o_w < o_r) \wedge \\ \qquad \bigwedge_{x \neq w \in W(v)} (o_x < o_w \vee o_r < o_x)\} \end{array} \tag{7}$$

Equation 8 gives a formula that enumerates three possible read-write relations for x_1^r and its corresponding writes in π_1.

$$\begin{array}{l} \{x_1^r = x_0^w \wedge o_0 < o_2 \wedge (o_{13} < o_0 \vee o_2 < o_{13})\} \vee \\ \{x_1^r = x_2^w \wedge o_{13} < o_2 \wedge (o_0 < o_{13} \vee o_2 < o_0)\} \vee \end{array} \tag{8}$$

DefUse Constraint ***(φ_{du})***. For a read v_r and a write v_w that do not form a DefUse pair in π, 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 π_1 given in Figure 4 is $\{W_0^z \rightharpoonup R_{13}^z, W_{13}^x \rightharpoonup R_2^x\}$.

Let Σ_π be the set of potential DefUse pairs in π. Equa-

tion 9 gives the formula for DefUse constraint. Partial DefUse constraint for π_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 $\Gamma^{im}_{\pi_1} = \{W^z_0 \rightarrow R^z_{13}, W^x_{13} \rightarrow R^x_2\}$. Then we project $\Gamma^{im}_{\pi_1}$ to the branch pair $\Sigma^{im}_{\pi_1} = \{b^F_{10} \Rightarrow b^T_1\}$, which is also removed from Σ.

$$\varphi_{du}[W^v \rightharpoonup R^v] = (o_w < o_r) \wedge \bigwedge_{x \neq w \in W(v)} (o_x < o_w \vee o_r < o_x) \quad (9)$$

$$\begin{array}{l} \varphi_{du}[W^z_0 \rightharpoonup R^z_{13}] : \ o_0 < o_{13} \wedge (o_{13} < o_2 \vee o_2 < o_0) \\ \varphi_{du}[W^x_{13} \rightharpoonup R^x_2] : \ o_{13} < o_2 \wedge (o_0 < o_{13} \vee o_2 < o_0) \end{array} \quad (10)$$

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 $\pi_B = \{b^{T/F}_1 \dots b^{T/F}_k\}$ be the set of branches in an explored path π. For any π' that is inferred from π by predicate analysis, we have $\pi'_B = \pi_B$, even though the orders of the branches in π and π' are different. In addition, for any π' such that $\pi'_B = \pi_B$, π' has the same set of instructions as π does. 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 $\pi = \langle \dots b^{T/F}_p \dots b^{T/F}_q \dots b^{T/F}_k \dots \rangle$ be an executed trace with k branches. By reversing $b^{T/F}_p$ to $b^{F/T}_p$, we observe the set of statements $B^{F/T}_p$ dominated by $b^{F/T}_p$. If there are any potential DefUse pairs between the statements in $B^{F/T}_p$ and $B^{T/F}_q$, $b^{T/F}_q$ in π and $b^{F/T}_p$ forms an escort branch pair. Any execution following $\pi' = \langle \dots b^{F/T}_p \dots \rangle$ satisfies the third aforementioned property. By consider one branch in π at a time, we can obtain a set of Σ^A_π of such escort branch pairs derived from π. Consider $\pi_1 = \langle 1T, 2, 6F, 9, 10F, 13, 14F, 17 \rangle$ in the running example, reversing 1^T or 10^F leads to potential DefUse pairs with existing branches in π_1. As a result, we have $\Sigma^A_{\pi_1} = \{b^T_1 \Rightarrow b^T_{10}, b^T_{10} \Rightarrow b^T_1, b^F_1 \Rightarrow b^F_{10}, b^F_{10} \Rightarrow b^F_1\}$.

After the dynamic execution of π, WS_π 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_π. Let φ_{sm}, φ_{po}, and φ_{im} be the semantics constraint, partial-order constraint, and interleaving matching constraint obtained from predicative analysis on π. We need to remove the branch terms from φ_{sm} because we need to change the constraint on branch. Besides, we need to remove input values from φ_{sm} in order to computing new inputs. Let φ'_{sm} be the revised semantics constraint without branch terms and input values. Given $b^{F/T}_p \Rightarrow b^{T/F}_q \in WS_\pi$, without loss of generality, we assume it is $b^{F/T}_p$ that has been negated in π. The encoding that enforces a path that takes $b^{F/T}_p$ is as the following:

$$\varphi_{b_p} = \varphi'_{sm} \wedge \varphi_{po} \wedge \varphi_{im} \wedge b^{F/T}_p \wedge \bigwedge_{1 \leq q \leq k}^{q \neq p} (o_q < o_p \rightarrow b^{T/F}_q), \quad (11)$$

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

Back to the example in Figure 2, $WS_{\pi_1} = \Sigma^A_{\pi_1} \bigcap \Sigma = \{b^T_1 \Rightarrow b^T_{10}, b^F_1 \Rightarrow b^F_{10}, b^F_{10} \Rightarrow b^F_1\}$. By projecting WS_{π_1} to branches, we can confirm there are two to-be-negated branches $b_1(b^T_1 \rightarrow b^F_1)$ and $b_{10}(b^F_{10} \rightarrow b^T_{10})$. Then we build their negation constraint formulas and verify them one by one. For the negation $b^F_{10} \rightarrow b^T_{10}$ as shown in Equation 12, $\varphi_{b_p}[b^F_{10} \rightarrow b^T_{10}]$ is unsatisfiable since b_{10} can't be negated in π_1 at all. Therefore, we can't compute a feasible input/schedule vector pair. For the negation $b^T_1 \rightarrow b^F_1$(omitting its equation), one input/schedule vector pair $(\langle 1, 2, 3 \rangle, \langle 1F \rangle)$ is computed from its satisfiable solution and leads to the new execution that covers the branch pairs $b^F_1 \Rightarrow b^F_{10}$ and $b^F_{10} \Rightarrow b^F_1$.

$$\varphi_{b_p}[b^F_{10} \rightarrow b^T_{10}] = \dots \wedge x^r_2 > 2 \wedge (o_1 < o_{10} \rightarrow a_0 > 1) \quad (12)$$

Algorithm 3 AltBranchComputation (Program P, Trace π, BranchPairSet Σ, InputScheduleSet Ω)

```
1: Let Σ^A_π be the set of escort branch pairs if one of the branch in π was reversed;
2: WS_π = Σ^A_π ⋂ Σ;
3: Let B = {b_1^{T/F}, b_2^{T/F}, ..., b_k^{T/F}} be the branch constraints in φ_π;
4: let φ'_sm be the semantics constraint of π without branch terms;
5: φ'_π = φ'_sm ∧ φ_po ∧ φ_im;
6: for each b_p^{F/T} ⇒ b_q^{T/F} ∈ WS_π do
7:    φ_{b_p} = φ'_π ∧ b_p^{F/T} ∧ ⋀_{1≤q≤k}^{q≠p} (o_q < o_p → b_q^{T/F})
8:    if φ_{b_p} is satisfiable then
9:       Obtain input α⃗ and thread schedule β⃗ from the solution to φ_{b_p}.
10:      Ω = Ω ∪ {(α⃗, β⃗)};
11:   end if
12: end for
```

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 Σ.

Algorithm 4 DefUseTesting (Program P)

```
1: Γ = Tested = ∅;
2: Σ = ProgramBranchPairs(P);
3: Ω = {⟨random value⟩, ⟨⟩};
4: while Σ ≠ ∅ ∧ Ω ≠ ∅ do
5:    π=GuidedExecution(P, Σ, Ω, Γ, Tested);
6:    if π = NULL then
7:       Continue;
8:    else
9:       PredicativeAnalysis(π, Γ);
10:      AlternateBranchComputation(P, π, Σ, Ω);
11:   end if
12: end while
13: return Γ;
```

- `GuideExecution` executes P under an input/schedule vector pair $(\vec{\alpha}, \vec{\beta})$ removed from Ω. The branch pairs covered by the executed trace π are removed from Σ, and the DefUse pairs discovered in π are added to Γ.
- `PredicativeAnalysis` conducts symbolic analysis on a given trace π. The computed DefUse pairs are added to Γ.
- `AlternateBranchComputation` computes branch pairs in P by negating one branch at a time in π. 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 $\Sigma = \{b_1^F \Rightarrow b_{10}^T, b_1^T \Rightarrow b_{10}^T, b_1^F \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^F, b_1^T \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^T\}$.
- Step 1: A random input and thread schedule $(\langle 2,2,3\rangle, \langle\rangle)$ leads to the execution of π_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 π_1 once its input and thread schedule are changed. The explicitly detected and implicitly computed DefUse pairs, eight in total, are recorded in Γ. The branch pairs covered by $\Gamma_{\pi_1}^{ex}$ and $\Gamma_{\pi_1}^{im}$ are $\Sigma_{\pi_1}^{ex} = \{b_1^T \Rightarrow b_{10}^F\}$ and $\Sigma_{\pi_1}^{im} = \{b_{10}^F \Rightarrow b_1^T\}$, respectively. We remove the items in $\Sigma_{\pi_1}^{ex}$ and $\Sigma_{\pi_1}^{im}$ from Σ, which gets $\Sigma = \{b_1^F \Rightarrow b_{10}^T, b_1^T \Rightarrow b_{10}^T, b_1^F \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^F\}$. From π_1, we may obtain a different execution by reverting either b_1^F or b_{10}^T, which means $\Sigma_{\pi_1}^A = \{b_1^T \Rightarrow b_{10}^T, b_{10}^T \Rightarrow b_1^T, b_1^F \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^F\}$. The alternate branch computation component produces one input/schedule vector in Ω that lead to the execution that cover the branch pair in WS_{π_1}. Note that since branch at Line 10 can't be negated in π_1, we only get one input/schedule vector that reverses branch at Line 1.
- Step 2: The removed item $(\langle 1,2,3\rangle, \langle 1F\rangle)$ from Ω causes the execution of π_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 Γ. Since π_2 covers $b_1^F \Rightarrow b_{10}^T$, such item is removed from Σ. If we revert the branches in π_2, we may cover $\Sigma_{\pi_2}^A = \{b_1^T \Rightarrow b_{10}^T, b_{10}^T \Rightarrow b_1^T, b_1^F \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^F\}$. Computing $\Sigma_{\pi_2}^A \bigcap \Sigma$, we get the to-be-covered branch pairs $WS_{\pi_2} = \{b_1^T \Rightarrow b_{10}^T, b_{10}^T \Rightarrow b_1^T, b_1^F \Rightarrow b_{10}^F\}$. The items lead to two new input/schedule vector pairs in Ω.
- Step 3: By following a removed input/schedule pair vector $(\langle 1,2,3\rangle, \langle 1F,4,6F,9,10F\rangle)$ from Ω we obtain π_3. The dynamic execution and predicative analysis detect 4 new DefUse pairs that are added to Γ. The branch pair covered by π_3 is $\{b_1^F \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^F\}$, which is removed from Σ. By reverting a branch in π_3, we may cover $\{b_1^T \Rightarrow b_{10}^F, b_{10}^F \Rightarrow b_1^T, b_1^F \Rightarrow b_{10}^T, b_{10}^T \Rightarrow b_1^F\}$. However, both have been covered before($WS_{\pi_3} = \emptyset$). Thus π_3 does not produces any to-be-covered paths.
- Step 4: The last execution $\pi_4 = \langle 1T,2,6F,9,10F,13,14F,17\rangle$ is determined by the remaining one item $(\langle 3,2,3\rangle, \langle 1T\rangle)$ in Ω. It equals π_1 and has been already analyzed or tested. So we ignore it and skip. However, since Ω are empty now, the algorithm terminates. We consider the items in Σ 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 Δ OF DEFUSE PAIRS.

Step	Component	Set Content
1	Guided Execution	$I:(\langle 2,2,3\rangle,\langle\rangle)\rightarrow\pi_1:\{1T,2,6F,9,10F,13,14F,17\}$(**New**)
		$\Gamma^{ex}_{\pi_1}=\{W^x_0\rightarrow R^x_2, W^x_0\rightarrow R^x_{10}, W^z_2\rightarrow R^z_{13}\}$
		$\Sigma^{ex}_{\pi_1}=\{b^T_1\Rightarrow b^F_{10}\}$
		$\Sigma=\Sigma-\Sigma^{ex}_{\pi_1}=\{b^F_1\Rightarrow b^T_{10}, b^T_1\Rightarrow b^T_{10}, b^F_1\Rightarrow b^F_{10}, b^F_{10}\Rightarrow b^F_1, b^F_{10}\Rightarrow b^T_1\}$
	Predicative Analysis	$\Gamma^{im}_{\pi_1}=\{W^x_{13}\rightarrow R^x_2, W^z_0\rightarrow R^z_{13}\}$
		$\Sigma^{im}_{\pi_1}=\{b^F_{10}\Rightarrow b^T_1\}$
		$\Sigma=\Sigma-\Sigma^{im}_{\pi_1}=\{b^F_1\Rightarrow b^T_{10}, b^T_1\Rightarrow b^T_{10}, b^F_1\Rightarrow b^F_{10}, b^F_{10}\Rightarrow b^F_1\}$
		$\Delta\Gamma=\{W^x_0\rightarrow R^x_2, W^x_0\rightarrow R^x_{10}, W^z_2\rightarrow R^z_{13}, W^x_{13}\rightarrow R^x_2, W^z_0\rightarrow R^z_{13}\}$
	Alternate Branch Computation	$\Sigma^A_{\pi_1}=\{b^T_1\Rightarrow b^T_{10}, b^T_{10}\Rightarrow b^T_1, b^F_1\Rightarrow b^F_{10}, b^F_{10}\Rightarrow b^F_1\}$
		$WS_{\pi_1}=\Sigma^A_{\pi_1}\bigcap\Sigma=\{b^T_1\Rightarrow b^T_{10}, b^F_1\Rightarrow b^F_{10}, b^F_{10}\Rightarrow b^F_1\}$
		$\Omega=\{(\langle 1,2,3\rangle,\langle 1F\rangle)\}$
2	Guided Execution	$I:(\langle 1,2,3\rangle,\langle 1F\rangle)\rightarrow\pi_2:\{1F,4,5,6F,9,10T,11,14F,17\}$(**New**)
		$\Gamma^{ex}_{\pi_2}=\{W^x_0\rightarrow R^x_4, W^z_4\rightarrow R^z_{11}, W^x_5\rightarrow R^x_{10}\}$
		$\Sigma^{ex}_{\pi_2}=\{b^F_1\Rightarrow b^T_{10}\}$
		$\Sigma=\Sigma-\Sigma^{ex}_{\pi_2}=\{b^T_1\Rightarrow b^T_{10}, b^F_1\Rightarrow b^F_{10}, b^F_{10}\Rightarrow b^F_1\}$
	Predicative Analysis	$\Gamma^{im}_{\pi_2}=\emptyset$
		$\Sigma^{im}_{\pi_2}=\emptyset$
		$\Sigma=\Sigma-\Sigma^{im}_{\pi_2}=\{b^T_1\Rightarrow b^T_{10}, b^F_1\Rightarrow b^F_{10}, b^F_{10}\Rightarrow b^F_1\}$
		$\Delta\Gamma=\{W^x_0\rightarrow R^x_4, W^z_4\rightarrow R^z_{11}, W^x_5\rightarrow R^x_{10}\}$
	Alternate Branch Computation	$\Sigma^A_{\pi_2}=\{b^T_1\Rightarrow b^T_{10}, b^T_{10}\Rightarrow b^T_1, b^F_1\Rightarrow b^F_{10}, b^F_{10}\Rightarrow b^F_1\}$
		$WS_{\pi_2}=\Sigma^A_{\pi_2}\bigcap\Sigma=\{b^T_1\Rightarrow b^T_{10}, b^F_{10}\Rightarrow b^F_1, b^F_1\Rightarrow b^F_{10}\}$
		$\Omega=\{(\langle 1,2,3\rangle,\langle 1F,4,6F,9,10F\rangle),(\langle 3,2,3\rangle,\langle 1T\rangle)\}$
3	Guided Execution	$I:(\langle 1,2,3\rangle,\langle 1F,4,6F,9,10F\rangle)\rightarrow\pi_3:\{1F,4,6F,9,10F,$ $5,13,14F,17\}$(**New**)
		$\Gamma^{ex}_{\pi_3}=\{W^x_0\rightarrow R^x_4, W^x_0\rightarrow R^x_{10}, W^z_4\rightarrow R^z_{13}\}$
		$\Sigma^{ex}_{\pi_3}=\{b^F_1\Rightarrow b^F_{10}\}$
		$\Sigma=\Sigma-\Sigma^{ex}_{\pi_3}=\{b^T_1\Rightarrow b^T_{10}, b^F_{10}\Rightarrow b^F_1\}$
	Predicative Analysis	$\Gamma^{im}_{\pi_3}=\{W^x_{13}\rightarrow R^x_4\}$
		$\Sigma^{im}_{\pi_3}=\{b^F_{10}\Rightarrow b^F_1\}$
		$\Sigma=\Sigma-\Sigma^{im}_{\pi_3}=\{b^T_1\Rightarrow b^T_{10}\}$
		$\Delta\Gamma=\{W^x_{13}\rightarrow R^x_4, W^z_4\rightarrow R^z_{13}\}$
	Alternate Branch Computation	$\Sigma^A_{\pi_3}=\{b^T_1\Rightarrow b^F_{10}, b^F_{10}\Rightarrow b^T_1, b^F_1\Rightarrow b^T_{10}, b^T_{10}\Rightarrow b^F_1\}$
		$WS_{\pi_3}=\Sigma^N_{\pi_3}\bigcap\Sigma=\emptyset$
		$\Omega=\{(\langle 3,2,3\rangle,\langle 1T\rangle)\}$
4	Guided Execution	$I:(\langle 3,2,3\rangle,\langle 1T\rangle)\rightarrow\pi_4:\{1T,2,6F,9,10F,13,14F,17\}$(**Tested**)
		$\Sigma=\{b^T_1\Rightarrow b^T_{10}\}$
	Alternate Branch Computation	$\Omega=\emptyset$(**Stoped**)

TABLE II
EXPERIMENTAL RESULTS

Program	LOC	#Inst	#F	#Tr	#BP	#DU	#imDU	#inDU	solTime(s)	Time(s)	Mem(m)	BT
account_bug	50	338	187	1	0	8	3	4	0.06	0.10	20	●
arithmetic_bug	84	1,410	2,900	2	0	16	3	12	0.35	0.38	17	●
banking_bug	217	3,788	1,536	1	23	27	10	21	0.78	0.82	27	●
twostage_10_bug	114	518	4,388	1	0	7	6	0	0.22	0.24	14	●
twostage_3_bug	114	669	231	1	0	6	5	1	0.08	0.09	8	●
lazy_bug	46	168	124	1	0	7	4	0	0.02	0.04	8	●
micro_5_bug	53	2,051	17,239	1	0	324	303	17	482.13	482.29	519	●
queue_bug	153	3,436	2,489	3	10	27	3	18	0.35	0.40	19	○
recoder_20_bug	85	893	13,033	1	8	6	6	0	0.35	0.37	18	●
recoder_2_bug	85	146	136	1	5	6	3	0	0.03	0.05	20	●
stack_bug	111	5,443	6,181	4	34	12	2	5	3.72	3.77	26	○
stateful_bug	60	1,239	1,244	1	0	8	4	3	0.12	0.13	10	●
token_ring_bug	54	658	456	3	19	16	6	4	0.18	0.20	28	○
blackscholes_bug	578	33,144	3,049	1	0	18	0	18	64.71	64.99	537	×
lu-c_bug1	1,386	29,575	8,274	2	604	65	3	136	1,585.60	1,586.10	1,675	●
lu-c_bug2	1,386	46,440	6,804	2	530	63	5	98	1,082.34	1,082.93	1,662	●
fft_bug1	1,465	42,201	13,096	2	395	110	3	257	2,363.99	2,366.63	3,289	●
fft_bug2	1,466	121,896	11,296	2	321	116	13	200	1,750.44	1,753.27	2,914	●
radix_bug1	1,534	80,583	39,608	4	1075	93	6	176	2,033.29	2,034.99	801	●
radix_bug2	1,537	78,645	15,906	3	719	86	8	106	362.74	364.10	543	●
lu-nc_bug1	1,155	129,449	20,623	1	499	63	10	88	5130.89	5,131.84	4,680	●
pfscan_bug	904	110,143	9,386	4	17	35	12	14	8.79	10.12	233	●
Average	574	31,492	8,099	2	194	51	19	54	675.96	676.54	774	-

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, including *fft, radix, lu-c*, *lu-nc*, *blackscholes*, and *pfscan*. If a program has many implicit DefUse pairs, it means that there exists hidden or unobserved behavior between threads, which makes 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 execute 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 ExceptioNULL, a constraint-solving based tool to predict interleavings that are likely to cause null-pointer dereferences [52]. CLAP replays execution of multithreaded programs via solving a constraint model which is converted from logged execution trace [53]. J. Huang et. al. present an automated technique 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, U1301254, 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

[1] L. A. Clarke, "A program testing system," in *Proceedings of the 1976 Annual Conference*, ser. ACM '76, 1976, pp. 488–491.

[2] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *ESEC/FSE*, 2005, pp. 263–272.

[4] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[6] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: Symbolic Execution of Java Bytecode," in *ASE*. New York, NY, USA: ACM, 2010, pp. 179–180.

[7] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International*

journal on software tools for technology transfer, vol. 11, no. 4, pp. 339–353, 2009.

[8] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic Execution for Software Testing in Practice: Preliminary Assessment," in *ICSE*, 2011.

[9] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, "Assertion guided symbolic execution of multithreaded programs," in *ESEC/FSE*, 2015, pp. 854–865.

[10] S. Guo, M. Kusano, and C. Wang, "Conc-iSE: Incremental Symbolic Execution of Concurrent Software," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, 2016, pp. 531–542.

[11] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng, "Do I Use the Wrong Definition?: DeFuse: Definition-use Invariants for Detecting Concurrency and Sequential Bugs," in *OOPSLA*, 2010, pp. 160–174.

[12] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," in *ASPLOS*, 2008, pp. 329–339.

[13] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective Sampling for Lightweight Data-race Detection," *SIGPLAN Not.*, vol. 44, no. 6, pp. 134–143, Jun. 2009.

[14] C. Flanagan and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," in *PLDI*, 2009, pp. 121–133.

[15] Y. Cai and L. Cao, "Effective and Precise Dynamic Detection of Hidden Races for Java Programs," in *ESEC/FSE*. New York, NY, USA: ACM, 2015, pp. 450–461.

[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting Atomicity Violations via Access Interleaving Invariants," in *ASPLOS*, 2006, pp. 37–48.

[17] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, "Automated Type-based Analysis of Data Races and Atomicity," in *PPoPP*, 2005, pp. 83–94.

[18] M. Xu, R. Bodík, and M. D. Hill, "A Serializability Violation Detector for Shared-memory Server Programs," in *PLDI*, 2005, pp. 1–14.

[19] J. Yu and S. Narayanasamy, "A Case for an Interleaving Constrained Shared-memory Multi-processor," in *ISCA*, 2009, pp. 325–336.

[20] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. IEEE, 2004, pp. 75–86.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*. ACM, 1995, pp. 24–36.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, ser. PACT '08. ACM, 2008, pp. 72–81.

[24] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *ICSE*. ACM, 2011, pp. 331–340.

[25] N. Jalbert and K. Sen, "A trace simplification technique for effective debugging of concurrent programs," in *FSE*. ACM, 2010, pp. 57–66.

[26] C.-S. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path Coverage for Parallel Programs," in *ISSTA*. New York, NY, USA: ACM, 1998, pp. 153–162.

[27] M. J. Harrold and B. A. Malloy, "Data flow testing of parallelized code," in *Software Maintenance, 1992. Proceerdings., Conference on*, Nov 1992, pp. 272–281.

[28] Su, Ting and Fu, Zhoulai and Pu, Geguang and He, Jifeng and Su, Zhendong, "Combining symbolic execution and model checking for data flow testing," in *ICSE*, vol. 1. IEEE, 2015, pp. 654–665.

[29] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E Platform: Design, Implementation, and Applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, Feb. 2012.

[30] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel Symbolic Execution for Automated Real-world Software Testing," in *EuroSys*, 2011, pp. 183–198.

[31] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2Colic Testing," in *ESEC/FSE*, 2013, pp. 37–47.

[32] T. Bergan, D. Grossman, and L. Ceze, "Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts," in *OOPSLA*, 2014, pp. 491–506.

[33] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Post-conditioned symbolic execution," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–10.

[34] P. Godefroid, "Compositional Dynamic Test Generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07, 2007, pp. 47–54.

[35] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional May-must Program Analysis: Unleashing the Power of Alternation," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10, 2010, pp. 43–56.

[36] R. Majumdar and K. Sen, "Hybrid Concolic Testing," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, 2007, pp. 416–426.

[37] W. Visser, C. S. Păsăreanu, and R. Pelánek, "Test Input Generation for Java Containers Using State Matching," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06, 2006, pp. 37–48.

[38] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient State Merging in Symbolic Execution," *SIGPLAN Not.*, vol. 47, no. 6, pp. 193–204, Jun. 2012.

[39] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided Test Generation for Coverage Criteria," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10, 2010, pp. 1–10.

[40] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2016.

[41] K. L. McMillan, "Lazy Abstraction with Interpolants," in *Proceedings of the 18th International Conference on Computer Aided Verification*, ser. CAV'06, 2006, pp. 123–136.

[42] ——, "Lazy Annotation for Program Testing and Verification," in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, ser. CAV'10, 2010, pp. 104–118.

[43] D.-H. Chu and J. Jaffar, "A Complete Method for Symmetry Reduction in Safety Verification," in *Proceedings of the 24th International Conference on Computer Aided Verification*, ser. CAV'12, 2012, pp. 616–633.

[44] K. Sen, "Effective Random Testing of Concurrent Programs," in *ASE*, 2007, pp. 323–332.

[45] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs," in *ASPLOS*, 2010, pp. 167–178.

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

Violation Bugs from Their Hiding Places," in *ASPLOS*, 2009, pp. 25–36.

[47] M. Musuvathi and S. Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs," in *PLDI*, 2007, pp. 446–455.

[48] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A Coverage-Driven Testing Tool for Multithreaded Programs," in *OOPSLA*. New York, NY, USA: ACM, 2012, pp. 485–502.

[49] M. Samak and M. K. Ramanathan, "Synthesizing Tests for Detecting Atomicity Violations," in *ESEC/FSE*, 2015, pp. 131–142.

[50] Y. Cai and Z. Yang, "Radius aware probabilistic testing of deadlocks with guarantees," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 356–367. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970307

[51] Y. Cai, S. Wu, and W. K. Chan, "ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs," in *ICSE*. New York, NY, USA: ACM, 2014, pp. 491–502.

[52] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting Null-pointer Dereferences in Concurrent Programs," in *FSE*, 2012, pp. 47:1–47:11.

[53] J. Huang, C. Zhang, and J. Dolby, "CLAP: Recording Local Executions to Reproduce Concurrency Failures," in *PLDI*, 2013, pp. 141–152.

[54] J. Huang and L. Rauchwerger, "Finding Schedule-sensitive Branches," in *ESEC/FSE*. New York, NY, USA: ACM, 2015, pp. 439–449.