RECIPE: Relaxed Sound Predictive Analysis

***Abstract*—The abstract goes here.**

I. INTRODUCTION

In a multithreaded application, a data race occurs when two concurrent threads access the same memory location, such that (i) at least one of the accesses is write access, and (ii) no explicit mechanism is enforced to prevent the threads from simultaneous access to the location [9]. Race detection is important not only because races often reveal bugs in the implementation, but also as the basis of other techniques and analyses like atomicity checking [7], [8], [11], data-flow analysis [1] and record/replay [2], [6].

In this paper, we focus on *sound* race detection, whereby reported races are guaranteed to be real. This is the key requirement for adoption of the tool by developers [**?**]. Ensuring soundness is a difficult challenge, which mandates dynamic forms of analysis. Indeed, extensive research has been carried out on dynamic race detection. The general goal has been to derive constraints from a given execution trace on event reordering, and check for the remaining reorderings whether they disclose data races. The key question then becomes about constraint extraction.

Existing Approaches: Numerous techniques have been proposed to date for race detection, which either sacrifice soundness [4], [**?**], [**?**], [**?**] or have significant coverage limitations [**?**], [**?**], [**?**]. Recently *predictive analysis* has emerged as a promising alternative, whereby a single trace is considered, and races are discovered by permuting the execution schedule governing the trace [**?**], [**?**].

Predictive analysis guarantees soundness, and has the potential for high coverage, though current predictive analyses suffer from two major limitations:

1) All the permutations attempted by the analysis must preserve the flow dependencies exhibited by the original trace. As an example, if in the original trace thread t_1 reads a shared variable that was written by thread t_2, then the same must hold in the permuted trace.
2) The analysis cannot step outside the boundaries of the input trace, e.g. by exploring branches that were not followed in that trace.

Both of these constraints are a conservative means of ensuring soundness. In an empirical study we conducted, which we describe in Section VII, we found that these restrictions often lead to serious loss in coverage.

As an illustration, we refer to the trace in Figure 1, where the statements in red denote an unexplored branch outside the trace. Existing predictive analyses are unable to detect the race between lines `2` and `5` because of the dependence between lines `3` and `4`, which is a barrier to the needed reorderings.

```
          x = 0; y = 0; z = 3;
                z = 2;
-----------------------------------------
        T1                  T2
-----------------------------------------
 1: y = 3;
 2: x = 1;
 3: y = 5;
                    4: if (y > z)
                    5:   print(1/x);
                    6: else
                    7:   print(2/x);
```

Fig. 1. Example illustrating ordering constraints beyond synchronization primitives

A second race that is missed, as it involves the statements in red, is between lines `2` and `7`.

Our Approach: We describe a novel approach to predictive race detection, which we have implemented as the RECIPE analysis tool, that is able to relax both of these constraints. Compared to the state of the art [**?**], RECIPE is able to detect x1.5 more races with only the first restriction relaxed, and x2.5 more races with both restrictions relaxed.

The departure point of our approach is to consider the explicit values of shared memory locations instead of dependencies between trace events. The gain in coverage is twofold:

1) Reorderings that violate the original dependence structure but preserve the branching history become possible. The values read by a branching statement may change so long as the branch condition evaluates to its original truth value.
2) In addition, in certain cases we can guarantee soundness while directing execution toward branches that diverge from the original trace. This mandates that the effects of an unexplored branch can be modeled precisely, which holds frequently in practice because the original and new branchees often access the same set of shared memory locations.

Returning to the example in Figure 1, we demonstrate how value-based reasoning improves coverage. First, value-based reasoning reveals that trace [`1`, `4`, `5`, `2`, `3`] is feasible, since the first assignment, `y = 3`, satisfies the condition `y > z`. Hence the race is discovered. A second race, between lines `2` and `7`, is detected by negating the condition. This is possible because the effects of the `else` branch can be modeled precisely. Negation leads to an execution starting at line `4` (where `y` $\equiv$ `0`). The race between lines `2` and `7` then becomes visible.

Concretely, we achieve value-level (rather than dependence-level) granularity computationally via a unique encoding of the input execution trace as a constraint system. The constraints are then processed by a satisfiability checker. This gives us the flexibility to explore nontrivial constraints, such as forcing a context switch after the first assignment to `y` or negating the condition in Figure 1. Importantly, these manipulations are beyond what dependence-based reasoning can achieve, since this view of the trace is too conservative.

Contributions: This paper makes the following principal contributions:

II. Technical Overview

In this section, we walk the reader through a detailed technical description of our approach based on the example in Figure 1. As input, we assume (i) a program P as well as (ii) a trace of P recorded during a dynamic execution.

A. Preliminaries

To facilitate our presentation, we first introduce some terminologies used throughout this paper. At the high level, a trace is a sequence of events recorded during the observation run.

Event An event, $e =< t, id, inst, map >$, is a concrete representation that captures the details about the runtime execution of a static instruction $inst$.

- t refers to the thread that issues the event, denoted as t^e.
- id refers to the id associated with each event, denoted as i^e. The key property of id is *uniqueness*, i.e., any two events in the trace own different ids. The other important property of id is *monotonicity*, i.e., the events from the same thread should own the strictly increasing ids. Throughout this paper, unless otherwise specified, we use the index of an event in the trace as its id, which satisfies the above properties. Throughout this paper, we denote an event as e_{id} with the id as the subscript. We may use the terms e_{id} and id interchangeably given their one-to-one correspondence, e.g., t^{e_3} and t^3.
- $inst$ is the static instruction. The instructions are three-address instructions involving at most three operands, which modern compilers commonly support. Specifically, we are interested in the types of instructions listed in Table I. When the variable does not appear on the left hand of an equation, such as y in $x.f = y$, it may refer to a variable, a constant or event object creation expression $new(...)$. The bop stands for the binary operator, which may refer to $+, -, *, /, \%, \wedge, \vee$ in the assignment, or refer to $<, >, =, \wedge, \vee$ in the branch. The target of the branch event is not important in our scope, therefore, we may abbreviate the branch instruction as the boolean expression afterwards. The listed instructions suffice to represent all trace events of interest. This is because a concrete finite execution trace can be reduced to a straight-line loop-free call-free path program (argument passing and value return of the method call are modeled as assignments). This standard form of simplification preserves all the data-race-related information contained in the original trace.
- map is a mapping from live expressions at the current event to their value (and thus a partial mapping from expressions to values). Live expressions include program variables, object fields, boolean expressions, etc. We use the notation $[\![exp]\!]^e$ to retrieve the value associated with expression exp at e. Specially, we also store the heap location value $[\![\overline{x.f}]\!]^e$ for a field $x.f$. Note the difference between $[\![\overline{x.f}]\!]^e$ and $[\![x.f]\!]^e$, which represent the location and the value stored in it respectively. The location value $[\![\overline{x.f}]\!]^e$ is computed as a pair $([\![x]\!]^e, f)$.

```
s ::=
                     y = x.f    |  (heapr)
                     x.f = y    |  (heapw)
                 z = x bop y    |  (assign)
       if (x bop y) goto ...    |  (branch)
          lock(l) | unlock(l)   |  (sync)
fork(t) | join(t)| begin(t) | end(t)   (thread)
```

TABLE I
Language syntax

Trace We propose the projection operation and domains to facilitate the reasoning of the trace.

- projection operation "$|$": $\tau|t$ contains only the (ordered) events from the thread t; $\tau|l$ contains the events involving the location l; $\tau|i$ contains only the event of which the id equals i; $\tau| \leq e$ denotes the prefix of e, i.e., the events preceding the event e; $\tau| \geq e$ denotes the events after e; $\tau| >= e_1 \wedge <= e_2$ denotes the events between e_1 and e_2.
- sets: $\mathcal{T}$ denotes the set of threads involved; $\mathcal{L}$ denotes the set of memory locations involved; $\mathcal{SV}$ denotes the set of shared variables, which reference the shared locations, the shared location l can be judged by counting the number of threads in $\tau|l$; $\mathcal{E}$ denotes the set of events involved.

B. Constraint System

The predictive analysis takes the program and a trace as the input. Suppose it starts with the following trace of the program in Figure 1: $e_1 e_2 e_3 e_4 e_5$, where the line number is used as the event id (subscript). A potential race is between the pair, (e_2, e_5), as they access the shared variable x from different threads and one is a write.

Given the trace, the predictive analysis preserves the same event sequence for each thread, i.e., T_1 still executes $e_1 e_2 e_3$ and T_2 still executes $e_4 e_5$, while rescheduling the events from different threads so that the potential race pair can run concurrently, i.e., e_2 and e_5 run concurrently. Therefore, our goal is to compute a new schedule that witnesses a race, under the (both inter-thread and intra-thread) data flow constraints, control flow constraints and synchronization constraints. In the following, we explain only the necessary constraints (the full description of constraints is explained in Section **??**).

To model the schedule, we assign an order (integer) variable to each event, e.g., O_{e_1}. The event with smaller order variable is scheduled earlier. Also, we introduce symbols to denote shared accesses, e.g., W_x^{id} (R_x^{id}) denotes the write (read) of the shared variable x by the event e_{id}. The local accesses are not used in this example and therefore omitted.

Race Condition The potential race pair involves two accesses of the same shared variable from different threads, where one access is a write. Given any potential race pair, e.g., (e_2, e_5), it is a real race if and only if the two accesses can occur at the same time (race condition), which is captured by the race condition constraint

$$O_2 = O_5$$

. The race condition constraint— together with the other constraints — guarantees the feasibility of the predicted race if a solution is found for the overall constraint system.

Intra-thread Constraints The predictive analysis needs to preserve the same event sequence for each thread, which further requires

- **Control Flow Constraints** Each thread takes the same control flows (or branch decisions) to reproduce the events. For the running example (ignoring the statements in red), this yields:

$$R^4_{\mathtt{y}} > 2 \equiv \mathbf{true}$$

  That is, the value of variable y read at e_4 is greater than 2.
- **Intra-thread Order Constraints** The events should follow the same thread-local order as in the original trace. This constraint is imposed by the fact that the two runs share the same instruction sequence and take the same control flows. We yield the following formula:

$$O_1 < O_2 < O_3 \wedge O_4 < O_5$$

  That is, the three events from T_1 are totally ordered, and the branch event e_4 in T_2 executes before the event e_5 inside the branch body (We do not include e_7 as it is not in the trace).
- **Intra-thread Data Flow Constraints** The events should respect the data flow constraints imposed by each instruction. For our running example, we obtain:

$$W^0_{\mathtt{x}} = 0 \wedge W^0_{\mathtt{y}} = 0 \wedge W^0_{\mathtt{z}} = 3 \wedge W^1_{\mathtt{y}} = 3 \wedge W^2_{\mathtt{x}} = 1 \wedge W^3_{\mathtt{y}} = 5$$

  As an example, $W^3_{\mathtt{y}} = 5$ denotes that the value assigned to variable y by event e_3 is 5.

Indeed, this constraint is satisfied by the assignments to y both at line 1 and at line 3.

Inter-thread Data Flow Constraints The inter-thread data flow constraints capture what writes the read events read from and under what scheduling condition the data flow occurs. The resulting formula for our example is

$$\begin{aligned} & (R^4_{\mathtt{y}} = W^0_{\mathtt{y}} \wedge O_4 < O_1) \\ \bigvee \; & (R^4_{\mathtt{y}} = W^1_{\mathtt{y}} \wedge O_1 < O_4 < O_3) \\ \bigvee \; & (R^4_{\mathtt{y}} = W^3_{\mathtt{y}} \wedge O_3 < O_4) \end{aligned}$$

Notice, importantly, that the formula associates the data flow constraints with the scheduling order constraints. As an example, $(R^4_{\mathtt{y}} = W^1_{\mathtt{y}} \wedge O_1 < O_4 < O_3)$ means that the read e_4 reads from the write e_1, under the condition that e_4 happens after e_1 and no other writes (such as e_3) interleave them.

Unexplored Branches Beyond the encoding steps so far, which focus on the given trace, we can often encode constraints along unexplored branches. In our running example, this is essential to discover the race between lines 2 and 7.

The conflicting accesses in this case, determined based on static analysis of P, are expressed as $O_2 = O_7$. In addition, we negate the path condition, thereby obtaining $R^4_{\mathtt{y}} > 2 \equiv \mathbf{false}$ in place of $R^4_{\mathtt{y}} > 2 \equiv \mathbf{true}$. We also model the other necessary constraints such as the intra-thread order constraint $O_4 < O_7$.

Constraint Solving Having conjoined the formuals from the different encoding steps into (i) a global representation of all feasibility constraints (path, ordering, assignment and other constraints) and (ii) the requirement for a given race to occur (expressed as simultaneous execution of the conflicting accesses), we discharge the resulting formula to an off-the-shelf constraint solver, such as Z3 or Yices. If successful, the solver returns a solution for the specified constraints.

In particular, the solution discloses a feasible trace that gives rise to the race at hand. The trace is identified uniquely via the order enforced in the solution over the variables O_i, which represent scheduling order.

III. Relaxation of Flow Dependencies

The main goal of our analysis is to, given an original trace τ over the events $\mathcal{E}$, derive a trace τ' over the same set of events, which has a new scheduling π of the events (the order of events inside each thread should remain unchanged) and a new mapping θ from variables to values in the events. Our first relaxation comes from the insight that, we allow the variables to read different values than in the original trace even if such variables are used to determine the control flow, while existing approaches enforce them to read the same values as in the original trace. As a result, our relaxation allows more schedules, which are likely to expose more races.

Not every scheduling or every mapping θ guarantees a new feasible trace τ', therefore, we need to compute the scheduling and mapping that lead to feasible trace. We explain the technique details in the following.

A. SSA Form of the Trace

First of all, we require the SSA form of the trace, i.e., each variable in defined exactly once in the trace. Such requirement is a prerequisite for the computation of the mapping θ, as it can map each variable to only one value. Another side effect is that the def/use chains become explicit in the SSA form, which simplifies the following analysis steps.

RECIPE handles the local assignment and heap accesses differently.

- **Local Assignment** The SSA form for local assignment resembles the SSA form of static instructions in compiler optimization, except that the loops and recursions are fully resolved in a concrete trace and there is no need for the Phi node. More concretely, we replace the variable v defined in an event, as well as the following uses of the definition, to a new variable v^{id}, where id is the unique id of the event. The uniqueness of the id guarantees

x = 0; **y** = 0;

T_1	T_2
`1: s=0;`	
`2: for(i=1;i<3;i++)`	
`3:     s+=i;`	
`4:` **y** `= s;`	
`5:` **x** `= 1;`	
`6:` **y** `= 5;`	
	`7: if (`**y** `> 2)`
	`8:  print(`**x**`+1);`
	`9: else`
	`10:  print(`**x**`+2);`

Fig. 2. Running Example (shared variables are in bold font).

$Trace$	$SSA\ form\ of\ Trace$
0: **x**=0	0: W_x^0=0
1: **y**=0	1: W_y^1=0
2: s=0	2: s^2=0
3: i=1	3: i^3=1
4: i<3	4: i^3<3
5: s=s+i	5: s^5=s^2+i^3
6: i=2	6: i^6=2
7: i<3	7: i^6<3
8: s=s+i	8: s^8=s^5+i^6
9: i=3	9: i^9=3
10: i<3	10: i^9<3
11: **y** = s;	11: W_y^{11} = s^8;
12: **x** = 1;	12: W_x^{12} = 1;
13: **y** = 5;	13: W_y^{13} = 5;
14: **y** > 2	14: R_y^{14} > 2
15: tmp=**x**+1;	15: tmp=R_x^{15}+1;
16: print(tmp);	16: print(tmp);

Fig. 3. Trace

that no two events define the same variable, i.e., each variable is defined exactly once. Note that the uses of a definition can be computed easily by scanning the trace afterwards for the variables of the same name before the next redefinition.

- **Heap accesses** The accesses of the local heap locations and the accesses of shared locations behave differently, therefore, we introduce different SSA form for them.
  - **Local Heap Accesses** The accesses of local heap locations behave similarly to the local assignment as both the definition and uses belong to the single thread. Therefore we model them as local assignments: We introduce a fresh local variable $\mathtt{l}^{id}_{[\![o.f]\!]}$ to replace each definition and the corresponding uses of a field f of the local object o. An assumption here is the uses and the definition still refer to the same base object o in the predicted run, which is enforced through the constraint specification (Section **??**).
  - **Shared Heap Accesses** The accesses of shared locations are more complex. Each shared read may read from one of multiple writes that define the shared location, under different schedules. Given the write $x.f = y$ or the read $y = x.f$, we introduce two symbols, $W^{id}_{\mathtt{x.f}}$, which denotes the write access by the event id to the field f of the shared object referenced by x, and $R^{id}_{\mathtt{x.f}}$, which denotes the read access by the event id to the location $x.f$. Note that, $x.f$ in the subscript is interpreted as the location, denoted as a pair $([\![x]\!]^e, f)$, rather than the value stored in the location. Similar to the local heap accesses, an assumption for the correlation is that the writes and reads still refer to the same base object o in the predicted run, which we guarantee by specifying additional constraints (Section **??**).

Figure 3 exemplifies a trace and its SSA form, where the trace is generated from the program in Figure 2. Each label in Figure 3 on the left hand denotes the id of the event. As seen, event 5 uses the variable s^2 defined at event 2 and defines the variable s^5, which is used afterwards. The read/write of the shared variables are denoted in the special form, such as R_y^{14} and W_y^{13}. We use x, y to represent the location for simplicity.

B. Constraint System

Based on the SSA form of the trace, we build the constraints to compute a new trace with the new schedule and new mapping of variables to values. To model the schedule, we introduce the order variable O_e for each event e. Given two events e_i, e_j from two different threads, $O_{e_i} < O_{e_j}$ in a trace means e_i is scheduled before e_j. We omit the synchronization constraints purposely, which are well explained in all existing predictive analysis techniques [**?**], [3].

Race Condition Given any pair of events, e and e', that both access a common memory location l, we demand that

$$\begin{array}{ll} & t^e \neq t^{e'} \\ \bigwedge & (e\ writes\ l\ \vee e'\ writes\ l) \\ \bigwedge & O_e = O_{e'} \end{array}$$

That is, (i) events e and e' are executed by different threads, (ii) at least one of the events performs write access to l, and (iii) the events run concurrently. This is a direct logical encoding of the definition of a race condition. We refer to both e and e' as the racy event.

Intra-thread Constraints Given a racy pair under analysis, e and e', suppose e is after e' in the original trace τ, the predictive analysis reschedules the events in the prefix of e, i.e., prior to e, so that e and e' can run concurrently. A necessary condition applied to predictive analysis techniques is that each thread should follow the same event sequence prior to e as in the original run. We explain the rationale underlying this requirement in Section **??**, but now proceed to explain how to guarantee it using constraints. The preservation of the event sequence for each thread requires the intra-thread constraints:

- **Branch Decision Constraints** The branches in the prefix should take the same decisions to reproduce the same set of events. More formally, we require that

$$\bigwedge_{e_i \in \tau| \leq e \wedge inst^{e_i} = x<y.} x < y \equiv [\![x < y]\!]^{e_i}$$

where e_i is a branch event of which the boolean expression is $x < y$ (we use the binary operator $<$ for illustration) and $[\![x < y]\!]^{e_i}$ stores the evaluation result of the boolean expression in the original trace τ. The constraint specifies that the boolean expression in the derived trace should be evaluated to the same boolean value. Importantly, different from existing analyses [**?**], [3], we do not pose the requirement that the values flowing into branching statements remain the same, but suffice with the relaxed requirement that the evaluation of branching expressions remain the same. For example, suppose the branch event $x < y$ takes the value $1 < 2$ in the original trace, existing analyses require the same values for the variables x and y in the predicted run, while we allow the branch event to take other values of the variables, such as $3 < 4$.

- **Intra-thread Order Constraints** The events in the sequence should follow the same order as in the original trace. More formally, we require that

$$\forall e_i, e_j \in \tau. \quad \begin{array}{l} t^{e_i} \equiv t^{e_j}. \\ i^{e_i} < i^{e_j} \;\Rightarrow O_{e_i} < O_{e_j} \end{array}$$

The constraint specifies that the two events from the same thread should follow the same order as reflected by the ids of the events.

- **Assignment Constraints** The mapping of local variables should not contradict each instruction. More formally, we require that

$$\bigwedge_{e_i \in \tau| <= e \wedge type(e_i) = assign|heapr|heapw} inst^{e_i}$$

where $inst^{e_i}$, such as $x = y + z$ or $R_x^{id} = y$, captures the constraint over the values of the variables imposed by the instruction. For example, $x = y + z$ is not satisfied if the solver computes a mapping from the three variables to $3, 4, 5$ respectively. Again we consider only the events in the prefix $\tau|e$, where e is the racy event that happens later. Specially, for the instruction involving object creation, $x = new(...)$, we encode the object as its address, i.e., a constant integer returned by $System.identityHashcode(x)$ at runtime.

Rescheduling through Relaxation of Flow Dependencies We now move to the novel feature of RECIPE, which is its ability to explore execution schedules that depart from the value flow exhibited in the original trace. More precisely, RECIPE is able to relax value flow dependencies in the original trace, whereby a thread reads a different value possibly written by other events (or even events from other threads) to the shared memory location, while enforcing feasibility. This is strictly beyond the coverage potential of existing predictive analyses, which restrict trace transformations to ones where any read access to a shared memory location must correspond to the same written values as in the original trace, which are usually from the same write events.

T_1	T_2
`1: x1 = new();`	
`2: x2 = new();`	
`3: y = x1;`	
`4: y = x2;`	
`5: x2.f = 5;`	
	`6: z = y;`
	`7: w = z.f;`

Fig. 4. Example illustrating the need to account for heap accesses during trace transformation, where the labelled number denotes the event id.

To ensure feasibility under relaxation of flow dependencies, we need to secure the flow between the read/write with the execution schedule. For example, in Figure 1, for the read access to y at line `4` obtains the value assigned to y at line `1`, we need the schedule where thread T_1 executes line `1`, then T_2 executes line `4`, and then the schedule switches again to T_1 to execute line `3`.

In general, the constraint formula, given read R_ℓ of location ℓ as part of event e with set $\mathcal{W}$ of matching write events (i.e., events including write access to ℓ), takes the following form:

$$\bigvee_{e_w \in \mathcal{W}} \begin{array}{l} (R_\ell^e = W_\ell^{e_w}) \\ \wedge \quad O(e_w) < O(e) \\ \bigwedge_{e' \in \mathcal{W} \setminus \{e_w\}} (O(e') < O(e_w) \vee O(e) < O(e')) \end{array}$$

This disjunctive formula iterates over all matching write events, and demands for each that (i) it occurs prior to the read event ($O(e_w) < O(e)$) and (ii) all other write events either occur before ($O(e') < O(e_w)$) it or after the read event ($O(e) < O(e')$).

An important concern that arises due to relaxation of flow dependencies is that heap accesses may change their meaning, i.e., they involve different base objects and do no longer match with each other. As an illustration, we refer to Figure 4. While the read at event `7` appears to match the write at event `5`, this is conditioned on the read at event `6` being linked to the assignment at event `4`. If the predicted run violates this link, then feasibility is no longer guaranteed. In particular, if reordering results in z being assigned the first rather than second allocated object, then the read at event `7` no longer matches the write at event `5`.

To address this challenge, we enhance the constraint system with the requirement that heap objects that are dereferenced in the following field reference before the candidate racing events retain their original address in the predicted trace. This achieves two guarantees: First, matching heap access events in the original trace are guaranteed to also match in the predicted trace. Second, candidate races in the original trace remain viable in the predicted trace as they still refer to same location. Note that, the resolution of the virtual method call is modeled as the branch events that check the type of the target object, for which the constraints naturally guarantee the branch to take the same decision, i.e., the call to be resolved to the same method implementation.

Formally, given pair e and e' of candidate racing events such that e happens after e' in the trace τ, we require that

$$\bigwedge_{e_i \equiv \mathtt{y=x.f} | x.f = y \wedge e_i \in \tau | e} x \equiv [\![x]\!]^{e_i}$$

This constraint fixes that all heap dereferences prior to the racy event retain their original base object as in the original trace τ.

By sending the above constraints to a solver, we compute the necessary schedule orders among the events as well as the mapping of the variables. The necessary schedule orders define a partial order among the events, which permit a set of schedules that define the complete order that complies with it.

IV. Exploration of Unexecuted Branches

We now switch to the second feature of RECIPE, which is its ability to reason about unexplored branches. At the high level, we simply need to derive a trace that contains the unexplored branch, after that, we can reduce the analysis to the analysis in Section **??** (starting from the SSA processing). RECIPE derives such traces by symbolically executing the unexplored branches to record the events and including these events in the new traces.

As shown in the following, suppose the symbolic execution $symEngine(\tau, e)$ returns the single-thread traces starting from the branch e, which now takes a different decision. The resultant trace is computed by first removing the events thread-locally after e (inclusively) and then appending one of the single-thread traces that represents a path in the unexplored branch of e. The negation $neg(e)$ of e, which is the same as e except the evaluation result of the boolean expression differs, is also appended.

```
   for e : τ do
       if type(e) = branch then
3:         S = symEngine(τ, e)
           for τ' : S do
               τ_r = τ − ((τ|t^e)| >= e)
6:             τ_r = τ_r.replace(e, neg(e) + τ')
           end for
       end if
9: end for
```

Symbolic Execution Our symbolic execution is realized on top of the dataflow analysis, which generates events for each instruction within the unexplored branch. The symbolic execution has known limitations in reasoning about the schedules and loops (and recursion). Therefore, our analysis assumes all the events in the unexplored branch happen atomically at the branch event e without interleavings from other threads, as illustrated also in the above algorithm (line 6). In this way, our analysis adopts the sequential reasoning. RECIPE will reschedule the events based on the relaxation. As for loop (or recursion), the analysis terminates the current data flow immediately after one iteration, i.e., we expand the loop for only once. Otherwise, the analysis terminates the data flow normally if the flow goes out of the scope of the unexplored branch.

The analysis starts with the event e (exclusively), with the current state σ at e as the initial state. Remember that we use the runtime values for two types of events: the heap value of the base objects and the evaluation result of the boolean expression in the branch. Other values are maintained but not used. Therefore, we also need to maintain the heap value and the boolean expression value in the symbolic execution.

First, we propose standard kill/gen rules for three basic heap instructions to maintain the heap update along a given path. Specially, we separate the object creation from the local assignment. In the object creation, we assign a unique integer to represent the newly created object and store it into the state σ. For field reference, $\sigma[x.f]$ is actually short for two-level interpretation $\sigma[\sigma[x].f]$, where the first finds out the base object $\sigma[x]$ and the second searches for the object referenced by its field f. The kill/gen rules take effect only when $x.f$ is not of the primitive type.

The analysis also generates the events, where the heap values are stored in the map of the event for latter use. For the branch, we store its boolean evaluation result. As the boolean result is available only after the branch takes the decision, we delay the generation of the branch event until the first event after the branch.

V. Soundness Proof

In this section, we prove formally the soundness of RECIPE.

Researchers [3], [10] propose two axioms, *prefix closedness* and *local determinism*, which we adapt to establish our feasibility guarantee. Suppose $\mathcal{F}$ denotes the domain of feasible traces.

Intuitively, Prefix closedness means that if a trace is feasible, then the trace is still feasible after we discard the events after a point. The underlying reason is that the events after a point cannot affect the events before it. More formally,

Axiom 1 (Prefix closedness)**.** *If $t_1\ t_2 \in \mathcal{F}$, then $t_1 \in \mathcal{F}$.*

Intuitively, local determinism says that only the previous events of the same thread determine the existence of an event. For example, the existence of an event is only determined by the branch thread-locally. However, the value read by an event may be affected by another thread because a shared read reads from the most recent write, which may come from a different thread. The value in other events, such as branch or write, are locally determined too. More formally,

Axiom 2 (Local determinism)**.** *Assume $t_1 e_1, t_2 \in \mathcal{F}$, and $t_1|thread(e_1) \approx t_2|thread(e_1)$, where $\approx$ denotes the two traces are equal if the data values in the read and write events are ignored, Each event is determined by the previous events in the same thread. There are four cases:*

- **Branch** . *If op(e_1)=branch, and e_1 is in the form of $x < y$ (without loss of generality), then there exists values v_x, v_y such that $t_2 e_1[v_x/data_x, v_y/data_y] \in \mathcal{F}$. Here $e_1[v_x/data_x]$ represents that we replace the value of x from $data_x$ to v_x in e_1.*

Operation	Kill	Gen
y=$new(...)$	$\sigma[y] \mapsto \star$	$\sigma[y] \mapsto newI, < t, cnt, y = newI, \bot >$
y=$x.f$	$\sigma[y] \mapsto \star$	$\sigma[y] \mapsto \sigma[x.f], < t, cnt, y = x.f, [\![x]\!] = \sigma[x] >$
$x.f = y$	$\sigma[x.f] \mapsto \star$	$\sigma[x.f] \mapsto \sigma[y], < t, cnt, x.f = y, [\![x]\!] = \sigma[x] >$
$z = x\ \ bop\ \ y$		$< t, cnt, z = x\ bop\ y, \bot >$
$if(x < y)$		$< t, cnt, if(x < y), [\![x < y]\!] = bool >$

TABLE II
DATAFLOW ANALYSIS

- **Read** . *If op(e_1)=read, and e_2 is a read event that is identical to e_1 except that it may read a different value, and t_2e_2 is consistent, then $t_2e_2 \in \mathcal{F}$.*
- **Write** . *If op(e_1)=write, and there is a value v such that $t_2e_1[v/data] \in \mathcal{F}$.*
- **Others** . *If op(e_1) is of type different from above and t_2e_1 is consistent, then $t_2e_1 \in \mathcal{F}$.*

Here, the consistency is defined as follows. A trace is consistency iff it satisfies (1) read-write consistency, i.e., read event should contain the value written by the most recent write event, and (2) the synchronization consistency, e.g., the lock acquire and release of a lock should not be interleaved by the lock operations of the same lock, the begin event of a thread should follow the fork event of the parent thread.

The local determinism resembles the version in [3], [10]. A crucial difference is that, for the branch event, we do not require all reads thread locally before it to read the identical values as in the original trace t_1e_1. Instead, we allow the previous reads to read different values as long as the branch event is evaluated to the same boolean value, which guarantees the existence of the following events. Such relaxation allows us to find more races, while still preserving the feasibility.

An important assumption of ours is, we assume all local accesses are recorded in the trace so that we can determine the boolean status of a branch event computationally. [3], which assume the local accesses are not recorded and cannot determine the status computationally, have to enforce the reads before a branch to read the same values, leading to a weaker feasibility criterion. We believe our axiom captures the real-world scenario more faithfully. Another important assumption is for the read-write consistency, i.e., we assume the heap invariant, so that we know exactly which reads correspond to which writes in the predicted run.

They are called axioms because they are the rules that the sequentially consistent system [10] should follow. We refer the readers to the detailed discussion [10], [3].

Theorem 1 (Soundness)**.** *All traces in the closure $closure(t, \Phi)$, which includes t and is closed under the derivations in Axioms 1 2, are feasible.*

Proof. We order the traces in $closure(t)$ as, $t_0 = t, t_1, t_2, t_3, \dots t_n$. Clearly, t_0 is feasible. In the following, we prove by induction, i.e., assume t_n is feasible, then t_{n+1} which is derived from $S = \{t_0, \dots t_n\}$ should be feasible. There are several possible derivations.

- if t_{n+1} is a prefix of $t' \in S$, then $t_{n+1} \in \mathcal{F}$ because of the prefix closedness. t_{n+1} shares the same mappings as t'.
- if t_{n+1} is derived from $t^1e^1, t^2 \in \mathcal{F}$, we have $t^1|thread(e^1) \approx t^2|thread(e^1)$.
  - $op(e^1) = branch$ and $e^1 = x < y$. Our solver computes a mapping θ^d of all relevant variables such that $\theta(x < y) = \theta^d(x < y)$, then $t^2e_1[\theta^d(x)/\theta(x), \theta^d(y)/\theta(y)] \in \mathcal{F}$. Note that the mapping computed by solver may be different from the mapping in the trace t^1.
  - $op(e^1) = read$. Our solver ensures the read-write consistency, i.e., t^2e^2 is consistent, then $t^2e^2 \in \mathcal{F}$. Note that e^2 is the same as e^1 except that it may read a different value.
  - $op(e^1) = write$. There exists a value v such that $t^2e^2[v/data] \in \mathcal{F}$, where v is included in the mapping computed by the solver.
  - $op(e^1)$ is of other types. Our solver ensures the consistency, i.e., t^2e^1 is consistent, then $t^2e^2 \in \mathcal{F}$.

□

Theorem 2 (Maximality)**.** $\forall t', s.t., t'|th \approx t|th for any thread th, if t' \notin closure(t, \Phi), t' is infeasible.$

Proof. We sketch the proof. All traces that satisfy $t'|th \approx t|th for any thread th$ □

VI. RELATED WORK

In this section, we survey related research on race detection with special emphasis on predictive trace analysis (PTA).

A. Race Detection Techniques: Broad Survey

Initial attempts to address the challenge of race detection focused on built-in synchronization primitives [**?**], [**?**]. These include locks as well as the wait/notify and start/join scheduling controls. Notable among these efforts is *lockset* analysis, which considers only locks [**?**]. Because the derived constraints are partial, permitting certain infeasible event reorderings, lockset analysis cannot guarantee soundness [**?**].

A different tradeoff is struck by the *happens-before (HB)* approach [5]. In this style of analysis, all synchronization primitives are accounted for, though reordering constraints are conservative. As an example, HB inhibits reordering of two synchronized blocks governed by the same lock. PTA is an instance of the HB approach, whose starting point is a concrete execution trace.

Recently there have been successful attempts to relax HB constraints. Among these are hybrid analysis [**?**], which permits both orderings of lock-synchronized blocks, as well as the *Universal Causal Graph (UCG)* representation [4], which also enables both orderings but only if these are consistent with wait/notify- and start/join-induced constraints.

B. Predictive Trace Analysis

Given concurrent execution trace t, a *maximal and sound causal model* based on t defines the set of all traces that a program that is able to generate t can generate (maximality), and only those traces (soundness). PTA is founded on the notion of sound causality, as it considers feasible reorderings of the input trace that prove a candidate data race as such.

The first to propose PTA are Smagardakis et al. [**?**]. In their original work, both synchronization constraints and inter-thread dependencies are preserved, where inter-thread dependencies are respected by only allowing reorderings that leave the dependence structure exhibited by the original trace in tact. This ensures that the values of shared memory locations remain the same, which secures the soundness argument, though maximality is not guaranteed.

Said et al. [**?**] describe a PTA that is also sound albeit not maximal. Similarly to our analysis, Said et al. perform symbolic analysis of the input trace, and then utilize an SMT solver to search for interleaved schedules that establish the presence of data races. Soundsness is guaranteed by their ability to precisely encode the semantics of sequential consistency. ExceptionNULL [**?**] is another example of a sound PTA without maximality guarantees, where the goal is to detect null dereferences rather than data races.

Serbanuta et al. [**?**] have recently shown that it is possible to build a model that is not only sound but also maximal: Any extension of the model with a new trace renders the model unsound. This provides a foundation for different forms of PTA, including data races, but also atomicity, serializability and other properties.

Inspired by this result and closer to RECIPE is the PTA technique of Huang et al. [**?**], which is both sound and maximal. This technique, too, makes use of an SMT solver based on symbolic encoding of the input trace. As part of the encoding, control-flow information is taken into account to enable reorderings that do not violate control dependencies. Still, the two relaxations that RECIPE features, which enable (i) value- rather than dependence-based reasoning about data flow and (ii) consideration of unexplored branches, are strictly outside the scope of Huang et al.'s technique. In Section VII, we demonstrate via direct comparison with Huang et al. the dramatic improvement in coverage thanks to these relaxation methods, which we prove to handle in a sound and maximal manner.

VII. EVALUATION

Our evaluation focuses on the effectiveness and scalability of our approach. To measure the effectiveness, we compare with the predictive analysis, RV []. We choose RV for two main reasons: (1) RV is the only open source predictive analysis, (2) RV represents the state-of-the-art approach, which is theoretically proven to have higher detection capability than other approaches.

Evaluation Method We conduct our experiment on a large set of applications, which are also used to evaluate RV. Specifically, the set includes large applications such as Jigsaw, Xalan, Lusearch. For the large applications that use the reflection, we adopt Tamiflex [] to to support the static analysis, which replaces the reflection calls with the concrete method calls recorded in the observation run. We omit the benchmark eclipse because the current version of Tamiflex leads to abnormal execution after the instrumentation, which throws exception during starting the main service. By applying our tool to such abnormal execution, we identify only 3 races, similar to the report of RV []. Besides, the benchmark montecarlo requires the input, which we downloaded from internet and simplified. For the large applications, we use the most lightweight configuration if possible.

As the detection capability depends on the observed run, for fair comparison, we monitor the execution once by recording all necessary information required by both approaches, and then apply both techniques to the monitored run. Besides, our reported data for RV may be different from the original report for two reasons: (1) the different observed runs lead to different set of races, (2) the original implementation of RV contains a bug in identification of the branches, which leads to the misses of many branches and the incorrect reduction of dependences during the analysis. We confirmed the bug with the author and fixed the bug in our experiments [1].

Our experiments and measurements were all conducted on an x86 64 Thinkpad W530 worksta- tion with eight 2.30GHz Intel Core i7-3610QM processors, 16GB of RAM and 6M caches. The workstation runs version 12.04 of the Ubuntu Linux distribution, and has the Sun 64-Bit 1.6.0_26 Java virtual machine (JVM) installed [2].

A. Effectiveness

Table III shows the main results of our analysis, which includes four sections: Trace (details about the trace), Races (detected races), Difference (comparison with RV) and Running time (the time taken by the analysis).

The Trace section includes the number of threads (Th), the number of shared reads ($Reads$) and shared writes ($Writes$), the subset of shared reads that read the base object references ($Base$), where the base object references are references used as the base/target in the following field reference or the method invocation, the number of branches (Br), the synchronization events ($Sync$) and the total number of events ($Total$), which includes local accesses in addition to the aforementioned events. Specifically, for the branch, we report it in the form of A/B, where A refers to the number of branches used by our analysis and B refers to the number of branches used by RV. To ensure that the predicted run sees the same base object at each shared read/write, RV inserts the artificial

[1] We report the bug in details with the test case https://sites.google.com/site/recipe3141/

[2] Note that Sun JDK 1.7 does not support the transformation of dacapo applications with tamiflex.

branch immediately in front of each shared access (and array accesses). We do not use such artificial branches.

We make interesting observations about the Trace section. The non-local events, i.e., all the events listed in Table III, occupy around 30% of the total trace in the first 7 benchmarks, but occupy less than 1% of the trace in the rest benchmarks, which have relatively more complex logic. The reads of the shared base objects occupy a small portion ($1/3$-$1/10$) of the total shared reads. The rest shared reads read only the primitive values or the references that are not the base references, e.g., the references involved only in the nullness check. Besides, our analysis involves significantly less branch events as compared to the RV approach. The difference plays an important role in the detection, which we will explain shortly.

The $Races$ section shows the number of races detected by Recipe-s, i.e., Recipe without exploring un-executed paths, Recipe, i.e., the fully-fledged version, and RV. By comparing the Recipe-s and Recipe, we find Recipe finds 100+ more races, which demonstrates the strength of exploring un-executed paths. Intuitively, Recipe-s predicts based on a single trace, while Recipe predicts based on multiple traces containing different execution paths. We also compare the Recipe version with RV version, as illustrated in the $Difference$ section, where $Diff$ shows the races found by Recipe but missed by RV, $Diff'$ shows the races found by RV but missed by Recipe.

First of all, Recipe finds many 150+ more races than RV. The reasons are multifold: (1) Recipe can reason about the accesses in the un-executed paths, while RV and Recipe-s can only reason about the accesses in the executed paths. (2) Recipe or Recipe-s allows the relaxation of the scheduling even if it breaks the inter-thread read/write dependence in the observed run. Recipe allows the majority of the shared reads, i.e., the reads of primitive values, to freely read from a different value from a different write as long as the value leads to the same branch decisions. RV, however, requires them to read the same values as in the observed run. (3) An critical optimization proposed by RV is to preserve dependences only for the reads before the preceding branches of the racy events, rather than all the reads. However, this optimization is underplayed by the fact that RV introduces huge amount of artificial branches, i.e., one before each shared field access, which ensures the use of the same base objects. We get rid of such artificial branches and instead, rely on the small amount of base read events to ensure the use of the same base objects (Section **??**). Our strategy reduces the number of branches greatly and amplifies the effectiveness of the optimization. We also conduct case studies (Section **??**) to better illustrate the scenarios.

Another interesting observation is, although Recipe should produce all races found by RV in theory, Recipe may miss races found by RV in practice (Column $Diff'$), i.e., Recipe is not strictly more effective than RV in practice. The underlying reason is due to the limits of the constraint solver: (1) the solver cannot compute constraints with very complex arithmatic operations (2) the solver does not support some program constants such as the scientific notation, $3E-10$.

The last section, Running time, compares the analysis time for both approaches. We find our approach is significantly slower than RV, e.g., RV often finishes within 200 seconds, while our approach may take more than 1 hour. This is because our approach needs to reason about the computation among variables inside the local access events, while RV needs to only reason about the order relations among the events.

Summary of advantage and weakness In general, our approach is flexible, which allows the relaxation of schedules and paths based on the fine-grained reasoning of value flows. As a result, it detects many more races compared to existing approaches. On the other hand, it is heavy weighted. The suggestion is to combine it with the lightweight approach with soundness guarantee such as RV, by treating RV as the preprocessing and instructing Recipe to skip those confirmed by RV. In this way, we can complement Recipe by finding those missed by it due to the limit of the constraint solver.

B. Case studies

We manually inspect the reported races in small applications to gain better understanding about our approach.

Relaxing the Inter-thread dependence Figure 5 demonstrates the relaxation of inter-thread dependence enabled by our approach. The code is from the benchmark bbuffer, where the line number is marked. Huang et al. [] detects the race between line 291 and line 400, but fails to detect the race between line 294 and line 400. The reason is as follows. In the observation run, the execution follows the order, lines 400, 291 and 294. For the event at line 294, its preceding branch at line 291 reads from line 400. Therefore, Huang et al. [] requires the predicted run to preserve the dependence between line 291 and line 400 so that line 291 reads exactly the same value and the branch takes the same branch decision. The dependence enforces the order constraint, $400 \rightarrow 291$, which further enforces the order $400 \rightarrow 291 \rightarrow 294$. Our approach does not require the existence of such dependence. Specifically, we allow line 291 to happen before line 400 in the predicted run as long as the value read by it leads to the same branch decision, which is true in this case. As a result, there is no order constraint between line 294 and line 400, and the two forms a race. We re-replay such race easily using the eclipse IDE breakpoints.

Relaxing the Paths Figure 6 demonstrates how our approach relaxes the schedulings to account for the unexecuted paths. Each thread invokes the method Sorting, which recursively starts two children threads if there are two more available entries in the thread pool (lines 8-11), or starts one child thread if there is only one available entry (lines 4-5). The availability is computed through the static method available with the constant `total`, which is equal to 5. The shared variable `alive` denotes the used entries.

Initially there is one sorting thread. After it starts Thread 1 and Thread 2, there are three threads alive and only two more entries are available. Suppose in the observation run, Thread 2 consumes both thread entries and starts the children

TABLE III
RELAXED ANALYSIS

	Trace							Races			Difference		Running time (sec)	
Benchmarks	Th	Reads	Writes	Base	Br	sync	Total	Recipe-s	Recipe	RV	Diff	Diff'	Recipe	RV
critical	3	13	7	5	2/13	6	78	8	8	8	0	0	8	2
airline	11	45	15	4	32/82	30	317	9	9	9	0	0	490	4
account	3	46	21	10	3/47	6	227	2	5	5	3	3	41	4
pingpong	4	7	7	3	0/15	6	111	1	1	1	0	0	19	1
bbuffer	4	640	118	10	634/1.1K	217	3.3K	13	25	9	21	5	62	5
bubblesort	26	1.3k	966	121	155/2.8K	322	8.4K	7	7	7	0	0	3295	3
bufwriter	5	165	52	75	16/130	44	525	4	10	2	8	0	63	9
mergesort	5	38	33	5	15/472	28	1.7K	3	10	3	10	0	37	5
raytracer	2	31	5	12	314/8,2K	676	94.5K	4	6	4	2	0	47	2
montecarlo	2	5	86	2	1.9K/38.2K	21.1K	1.9M	1	4	1	3	0	1	17
moldyn	2	605	61	104	19.6K/52.6K	62	203.4K	6	14	2	12	0	2842	1
ftpserver	28	684	299	71	4.4K/233.3K	78.2K	3.9M	99	152	57	108	13	811	153
jigsaw	12	525	702	211	63.2K/467.9K	86.7K	5.5M	17	23	8	15	0	33	7
sunflow	9	2.1K	1.3K	473	201.3K/827.0K	50K	7.1M	38	78	20	69	11	4520	22
xalan	9	1.4K	0.9K	209	15.7K/103.2K	190.1K	6.6M	2	6	2	4	0	5317	10
lusearch	10	2.3K	0.5K	715	22.2K/164K	93.2K	9.1M	27	49	14	38	5	5430	8

```
waitForBug(···)
{
291: if(_finished !=threshold)
      {···}
294  y=_finished;
}
```

```
run()//Consumer.class
{
400: _finished++;
}
```

Thread 1 Thread 2

Fig. 5. Relaxation of Inter-thread dependence

Thread 3 and Thread 4 (not shown), updating `alive` to 5, then Thread 1 cannot execute the branch at lines 4-5. Huang et al. [] require the predicted run to preserve the dependence denoted by the dotted line since the dependence affects the branch condition at line 2. As a result, the predicted run follows the same branch decision and cannot reason about the code at lines 4-5. Our analysis does not have such limitation. Instead, it allows the predicted run to reason about the unexplored code. Specifically, it does not need to preserve the dependence from line 11 to line 1 and it allows line 1 (Thread 1) to read from line 9 (Thread 2). As a result, the branch condition guarding the unexplored branch is evaluated to be true, enabling the unexplored path in the constraint solver. Finally, the solver identifies the race between line 5 (Thread 1) and line 1 (Thread 3). Note that the two lines are synchronized on different locks [3].

VIII. VALIDITY THREAT

Java method can have 65535 bytecode instructions maximally. Therefore, we count the number of bytecode instructions inside each method, if the number exceeds 65525, we avoid instrumenting the method. The consequence is that, we will miss the races inside the method. In this case, we specify the variables read from the method to be equal to their concrete values in the constraints. THe constraint solver can proceed safely without being affected by such methods.

[3]We abbreviate the `synchronized` keyword as `sync`.

```
static sync available(){   return total-alive; }
```

Thread 1

```
Sorting(...)
{
    // child1=...
1     y= available();
2     if(y==0){···}
3     else if(y==1){
4        child1.start();
5        sync(this){alive++;}
6     }
7     else{
8        child1.start();
9        sync(this){alive++;}
10       child2.start();
11       sync(this){alive++;} ...
      }

}
```

Thread 2

```
    ...
9   sync(this){alive++;}
    ...
11  sync(this){alive++;}
```

Thread 3

```
1   available();
```

Fig. 6. Relaxation of Paths

We do not support the boolean operations such as &, bit operators $<<$, which contributes to most of our misses.

IX. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 316–326, New York, NY, USA, 2008. ACM.

[2] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 207–216, New York, NY, USA, 2010. ACM.

[3] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, 2014.

[4] V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 434–449, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM '1987*.

[6] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 463–474, New York, NY, USA, 2012. ACM.

[7] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[8] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[9] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 27–37, New York, NY, USA, 1997. ACM.

[10] T. F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *RV*, 2012.

[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 *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 160–174, New York, NY, USA, 2010. ACM.