Bare Demo of IEEEtran.cls for ICSE'15 and Workshops

Michael Shell*, Homer Simpson†, James Kirk‡, Montgomery Scott‡ and Eldon Tyrell§
*School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, Georgia 30332–0250
Email: see http://www.michaelshell.org/contact.html
†Twentieth Century Fox, Springfield, USA
Email: homer@thesimpsons.com
‡Starfleet Academy, San Francisco, California 96678-2391
Telephone: (800) 555–1212, Fax: (888) 555–1212
§Tyrell Inc., 123 Replicant Street, Los Angeles, California 90210–4321

Abstract—**The abstract goes here.**

I. Introduction

In a multithreaded application, a data race occurs when two (or more) concurrent threads access a mutual memory location with neither ordering constraints nor guarding synchronization. This form of unstructured access to shared memory is often unintended, indicating a bug in the code.

A main challenge in detection of data races is that they are elusive, arising only in certain concurrent schedules. Because thread interleavings are decided nondeterministically by the scheduler, it is hard to anticipate and recreate data races. Indeed, in practice data races often manifest rarely, leading e.g. to situations where the bug is missed during testing but surfaces up during production runs (due to its rarity combined with differences in hardware, usage scenario, etc).

a) Existing Approaches: Various approaches have been proposed to detect data races, which strike different tradeoffs between (i) *soundness*, whereby no false races are reported to the user, and (ii) *completeness*, whereby no true races are missed. Sound techniques are largely based on, or refine, the *happens-before* (*HB*) relation [**?**]. Unsound techniques that opt for more complete detection are based on lockset analysis, as popularized by the Eraser tool [**?**].

Driven by the requirement for high usability, which is governed by low to zero user tolerance to false warnings [**?**], we focus on fully sound detection techniques. A recent result in this space is Smagardakis et al.'s *causally-precedes* (*CP*) relation, which soundly relaxes HB edges between critical sections that do not exhibit conflicting accesses. A refinement of the CP model, proposed by Huang et al. [**?**], is to factor in control-flow information: Inter-thread CP edges, such that statements succeeding the target statement are not control dependent on it, are soundly discarded.

Both of these techniques, as well as preceding HB approaches, ensure soundness via notions of *dependence*. As long as certain data and/or control dependencies are respected, reported data races are guaranteed to be real. While these are useful steps to detect more data races, the reported warnings are restricted to (i) the given execution trace and (ii) reorderings that preserve essential (data and/or control) dependencies.

```
1  public class Example {
2    static int x = −1;
3    static int y = −1;
4
5  public static void main(String[] args) {
6    MyThread t = new MyThread();
7    t.start ();
8    y = 3;
9    x = 0;
10   y = 5; }
11
12 static class MyThread extends Thread {
13   public void run() {
14     if (y >= 3) {
15       // Race: is it possible to print −1?
16       System.out.println(x);
17     } else {
18       System.out.println(x + "else");
19     } } } }
```

Fig. 1. Benchmark from Huang et al.'s suite [**?**] (with slight revisions) that exhibits two potential data races

b) Motivation: As we illustrate in Figure **??**, taken from the benchmark suite of Huang et al. [**?**], these restrictions can potentially block real races from the analysis' view. In the `main` method of class `Example`, a new thread `t` is spawned. Subsequently, and in parallel, variable `y` is both assigned by the main thread and queried by the spawned thread, which checks whether `y`'s value is at least 3.

There are three possibilities: Either (i) the test at line 14 is reached before `y` is first assigned within `main`, and so the initial value of `y` of -1 is used and the test evaluates to `false`, or (ii) the test occurs after the first assignment and succeeds for `y` $\equiv$ `3`, or (iii) the test takes place after the second assignment, again succeeding bu this time for `y` $\equiv$ `5`. In specific, because the test on `y` may either succeed or fail, both branches of the conditional block within the `run` method of inner class `MyThread` may execute. Similarly to the analysis for `y`, we notice that the first `println` statement can either print the initial value of `x` (-1) or its value after the assignment inside `main` (0). To conclude, there are two potential data races in the code.

The CP model is too restrictive to discover both of these races, even when control-flow information is factored into the analysis. To demonstrate this, let us consider trace $[6, 7, 8, 14, 16, 9, 10]$. That is, the first context switch occurs after `y` is first assigned, method `run` then executes to completion, and subsequently `main` finishes executing. There is genuine control dependence between the statements at lines 8 and 14, which disables any form of reordering according to CP, the result being that both races are missed. Indeed, if we restrict the analysis to the given trace and dependencies that arise within it, then the races in Figure **??** remain out of reach.

c) Our Approach: The main observation underlying our approach is that accounting for dependencies alone is insufficient. The departure point of this paper is in modeling the *values* of shared variables explicitly, rather than stopping at the level of *dependencies* within and between threads. Importantly, this enables us — in certain cases — to step outside the boundaries of the given trace and explore additional program behaviors.

Applied to the example in Figure **??**, our approach permits us to exploit two important facts:

- First, the test at line 14 is satisfied both after the first assignment and after the second assignment to `y`. In both cases $y \geq 3$. Making this observation, which refers directly to the values assumed by `y` and their use in control-flow decisions, lets us account for the assignment `x=0`, thereby uncovering the race over `x`.
- Second, and even more significantly, explicit modeling of values allows us to simulate a context switch before `y` is first assigned inside `main`. In this scenario, `y` has the value -1, which leads to a negative evaluation of the test at line 14. Note that we are no longer reordering events within the same trace, but have switched to another trace that is still guaranteed to be feasible since we are able to precisely maintain the program's state and behavior along all transitions (including those within the `else` branch, which was not executed in the input trace).

To summarize, relaxing the ability to reoder events — to the point of simulating new execution traces beyond the input trace — is the key to improving detection capabilities. The principal idea undetlying our approach is to model the shared state explicitly, rather than merely recording dependencies that arise within the given trace. As we demonstrate experimentally in Section **??**, this form of refinement has a dramatic effect on detection capabilities compared to previous approaches. ** MORE EXPERIMENTAL DATA NEEDED **

At the technical level, we represent the shared state, and ensure the feasibility of reorderings and execution along new branches, via a novel encoding of relevant events as a constraint system. ** MORE DETAILS NEEDED **

d) Contributions: The principal contributions of this paper are the following:

II. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.