User Evaluation of Correctness Conditions: A Case Study of Cooperability

Caitlin Sadowski Jaeheon Yi
University of California at Santa Cruz
{supertri, jaeheon}@cs.ucsc.edu

Abstract

In order to find and fix concurrency bugs, programmers must reason about different possible thread interleavings – context switches may occur at any program point, all with the potential for thread interference. To reduce the number of thread interference points to consider, the correctness criterion of cooperability ensures that code executes *as if* context switches may happen only at specific *yield* annotations. This paper provides empirical evidence that cooperability makes it easier to find concurrency bugs.

1. Background

Today, multicore processors are standard. Taking advantage of such processors entails using concurrent programs, where multiple threads of control work simultaneously. However, concurrent programs suffer from concurrency-specific errors [8]; programmers who write multithreaded code contend with *preemptive* scheduling, in which a context switch may occur at any program point. Many concurrency errors are a result of the difficulty in understanding the consequences of preemptive scheduling, which results in a combinatorial explosion of possible thread schedules and introduces scheduling-dependent bugs that are difficult to find, understand, and fix.

In contrast, *cooperative* scheduling permits a context switch only at an explicit *yield* annotation. Cooperative scheduling enables sequential reasoning between successive yield annotations but is inadequate as an execution model since it fails to exploit multicore hardware.

The correctness criterion of *cooperability* [17] combines these two scheduling semantics: a preemptively-scheduled execution trace is cooperable if it behaves equivalently to some cooperatively-scheduled trace. A program is cooperable if every preemptive-scheduled trace that the program could generate is cooperable. Thus, even though a cooperable program executes under preemptive scheduling, we can still reason about its correctness using cooperative scheduling. The set of yield annotations for a program is *correct* if it guarantees cooperability; we can check that such annotations are correct through static or dynamic analysis.

The example in Figure 1 illustrates these concepts. The `Ticket` class has a buggy `valuate()` method that is unsafe under different thread interleavings. The two yield annotations in the program represents all observable interference, and so the program is cooperable. The two traces represent program execution under preemptive and cooperative scheduling; their behavior is equivalent. The property of cooperability allows us to reason about the buggy behavior of trace (a) using the cooperative trace (b).

2. Problem Statement

Within the programming languages community, it is fairly common to claim a particular tool or methodology is "easier" without justification [15]. We would like to empirically evaluate the usefulness of cooperability for programmers:

Does cooperability help programmers identify bugs, compared to examining code without yield annotations?

We investigate three hypotheses:

- *H1: Participants have an easier time finding bugs in cooperable code,* i.e.*, with correct yield annotations.*
- *H2: Participants have a higher subjective impression of whether they understand code with correct yield annotations, as compared to code without such annotations.*
- *H3: Expert participants issue higher quality bug reports, as compared with novice participants.*

2.1 Contributions

In this paper, we contribute the following:

- We give the first user evaluation of a correctness condition.

```
1  class Ticket {
2    float value;
3    public void valuate() {
4      float tmp1 = value * 0.10;
5      /* yield; */
6      float tmp2 = value + tmp1;
7      /* yield; */
8      value = tmp2;
9    }
10 }
```

Thread 1	*Thread 2*
rd value	
	rd value
wr tmp1	
	wr tmp1
yield	
	yield
rd tmp1	
	rd tmp1
rd value	
	rd value
wr tmp2	
	wr tmp2
yield	
	yield
	rd tmp2
rd tmp2	
	wr value
wr value	

(a) A preemptive scheduling

Thread 1	*Thread 2*
rd value	
wr tmp1	
yield	
	rd value
	wr tmp1
	yield
rd tmp1	
rd value	
wr tmp2	
yield	
	rd tmp1
	rd value
	wr tmp2
	yield
	rd tmp2
	wr value
rd tmp2	
wr value	

(b) An equivalent cooperative scheduling

Figure 1. A cooperable program with two equivalent traces.

- We outline our hypotheses, methodology, and experimental design for evaluating the usefulness of cooperability.
- We report encouraging results of the usefulness of cooperability, discuss what we have learned, and provide several directions of future work.

3. Related Work

A programming language is, by definition, a form of interaction between humans and computers. One aim of programming languages research is to make it easier for people to write correct, scalable, understandable code. With this intent in mind, one might imagine the two subfields of human computer interaction (HCI) and programming languages (PL) share more research than they do. In the following subsection we will give an overview of some issues comparing usability of different programming languages, features, or systems. This overview is followed by a brief survey of literature on comparing parallel programming systems.

3.1 User Evaluation of Programming Languages

Research at the intersection of HCI and PL is becoming something of a hot topic [10], as increasing numbers of researchers are looking to HCI methods to help improve software engineering practices. Here, we focus on some prior research where multiple languages, tools, or programming environments are given a usability (*i.e.* productivity) comparison.

One challenge of comparing different languages or language features is the inherent difficulty in measuring productivity in a programming language. Most studies have participants write programs in the languages or systems under exploration. But which program do they write? Different languages, paradigms, or tools perform better on different code samples. Experiments where participants write programs are (necessarily) severely constrained in the size of and number of programs, and do not clearly extend to real-world scenarios involving large programs, legacy code, and different program styles.

Some studies focus on measuring the end result (*i.e.* the program) in terms of performance.There is substantial bias in the choice of a language or system to implement a given program, or in the choice of benchmarks given a particular framework [14]. We feel that it is important to measure correctness along with performance. However, checking correctness of programs is a vibrant area of research in the fields of systems, programming languages and software engineering. Doing so requires a clear checkable specification, regression test suite, model checking system, or similar setup. Measuring correctness of parallel programs is even more difficult [8].

A separate but related question is how much programmer effort is required to create a program (or perform some other programming-related task) with the language or system. Although several metrics exist for measuring programmer effort, the validity of existing metrics has been heavily criticized [9]. One popular metric for programmer effort in a particular language or system is the amount of time taken to complete a task (*e.g.* write a program); however, it is difficult to accurately identify all time spent on a specific task. Due to the difficulties with measuring programmer time, a number of indirect metrics for effort have been used. It is unclear how these metrics actually relate to program performance and programmer time, but they are relatively easy to measure and have been used in many studies. For example, the number of lines of code (LOC) has been used as a proxy for programmer effort under the assumption that shorter programs are faster to write [9] and also program correctness under the assumption that shorter programs contain less bugs [16]. Other indirect metrics involve counting things such as the number of compilations, the number of runs while developing the program, or the number of times the debugger is used [9, 16].

3.2 User Evaluation and Parallel Programming

Multithreaded and parallel code is difficult to write, understand, and debug [7]. Competing paradigms, languages, teaching styles, and tools have been debated in the research literature. Very little work has been done at the intersection of parallel programming and usability, and the small body of research that does exist in this area is mostly inconclusive [2, 4, 9, 16].

Szafron and Schaeffer compared a message passing library to a GUI-based parallel programming system called Enterprise [16]. The participants for this experiment were graduate students completing an assignment. The comparison was inconclusive; the messaging passing library implementations were faster and implemented with less login time to the development machines, but the Enterprise implementations had fewer LOC and fewer compiles.

One large study had 237 undergraduate students implement the same program with coarse- and fine-grained locks, monitors, and transactions [15]. They found that students consistently thought coarse- and fine-grained locking were easiest to think about and had the best syntax, even though their subjective impressions of transactional memory improved with experience.

3.3 User Evaluation and HPC

A growing group of researchers have been looking at programmer productivity for developing High Performance Computing (HPC) software. A collaborative of universities is currently conducting a variety of classroom studies to compare different programming models across a variety of representative applications [5].

Luff compared multithreading with the actor model (*i.e.* message passing), transactional memory, and a sequential control sample [9]. Each participant wrote multithreaded code and code that fit in one of the other cases. The goal of this work was to measure programmer effort, and the three main metrics used were time taken to write the program, LOC, and subjective impressions. The end result was inconclusive, although the learning effect was significant. The other interesting result was that there was a bimodal distribution where either participants finished their code by the two hour mark, or did not finish at all.

Hochstein *et al.* compared novice serial, OpenMP (shared memory), and MPI (message passing) implementations of two problems [5]. They found that MPI effort (in terms of time and LOC) was greater than OpenMP effort, but there were not enough data points to compare program performance in a statistically significant way. Hochstein *et al.* later compared the parallel random access memory (PRAM) model, which supports synchronous parallel operations to avoid common concurrency errors like data races, with MPI [6]. They found that the PRAM-like implementations were faster, but there was not a statistically significant different in program correctness.

Ebcioglu *et al.* compared three different languages, and found that the X10 language had a shorter development time: defined as time to either a correct solution or giving up on the problem [2]. However, the authors do not report if this result is statistically significant. What is particularly interesting about this study is that a large subset (more than 1/3) of the participants in each of the three groups did not successfully complete a correct solution that exhibited *any* speedup. This study reinforced that parallel programming is very difficult.

4. Methodology

Because this is the first user evaluation of a correctness condition, developing a robust methodology for the evaluation was challenging. We wished to minimize the time burden on the participants, and develop an evaluation that could be easily deployed at different locations. With these constraints in mind, we settled on using an internet-based survey to collect responses. Our goal was to develop a survey that would take approximately half an hour, but we experienced a wide variation in (self-reported) completion times, ranging from 15 to 90 minutes.

In the survey, we first collected information about the participants' background in multithreading and had participants answer a few questions about multithreading concepts to test their experience. We asked each participant to define the meaning of data races and atomicity violations, two common types of multithreaded code bugs. We used these questions as an additional check to ensure that participants had enough understanding of multithreaded code to successfully complete the survey.

We used a between-group study design, in which survey participants were randomly assigned to one of two groups: with-annotations or without-annotations. In order to measure whether yield annotations helped participants find bugs, the main body of the survey consisted of three code samples that each contained a concurrency bug. Half of the participants received samples with yield annotations. The other half received the *same* three code samples, but *without* annotations.

For each code sample, participants were asked to examine the code briefly, and provide some documentation for the code. They were then asked to identify all code bugs (with a particular emphasis on bugs caused by multithreading). After they identified code bugs for a sample, we asked them to estimate how long it had taken them to understand that sample, and then rate their confidence in both their overall understanding of the code and whether they had identified all code bugs. Afterwards, we asked participants in the with-annotation group whether they found the annotations helpful; the without-annotation group was not asked this question.

4.1 Examples

Our survey contains three code samples: `ArchivalList`, `IntList`, and `StringBuffer`.[1] To find realistic examples containing concurrency bugs, we first examined concurrency bug patterns found in the wild [8]. Unfortunately, these published bug examples need many supporting details to make them believably self-contained, without requiring knowledge of libraries or systems.

The `ArchivalList` example is based on a concurrency bug from the above paper; this sample involves a doubly-linked list (based on a thread-safe `Node` class) that has un-

[1] We also developed one optional example. Due to the low reporting rate, we do not discuss this optional question in the Results section.

```
1  public class StringBuffer {
2    private char[] value;
3    private int index;
4
5    public StringBuffer(int capacity) {
6      value = new char[capacity];
7    }
8
9    public synchronized StringBuffer append(
10       StringBuffer sb){
11     if (sb == null)
12       return this;
13     int len = sb.length();
14     /* yield; */
15     int newlen = len + index;
16     if (newlen > value.length)
17       expandCapacity(newlen);
18     sb.getChars(0, len, value, index);
19     return this;
20   }
21
22   private synchronized void getChars(
23       int beg, int end, char[] dst, int dBeg) {
24     System.arraycopy(value, beg, dst, dBeg, end-beg);
25   }
26
27   private synchronized void expandCapacity(
28       int newlen) {
29     char[] tmp = new char[newlen];
30     System.arraycopy(value, 0, tmp, 0, value.length);
31     value = tmp;
32   }
33
34   public synchronized int length() {
35     return index;
36   }
37 }
```

Figure 2. String Buffer Example.

synchronized code to change the pointer to the last element in the list.

We then examined papers on analysis tools which found concurrency bugs for more self-contained realistic code samples. The `IntList` example is based on a bug found by the AtomFuzzer atomicity violation detector [13].

The `StringBuffer` example (Figure 2) contains an atomicity violation from the Java libraries [3]. In this figure, the `/*yield;*/` annotation represents the fact that another thread could change the contents and length of `sb` at this program point, for example, by executing `sb.append()`. Note that yield annotations do not change code behaviour, they just document the points of observable thread interference.

4.2 Participants

We advertised the user study through a school of engineering graduate mailing list, on a gaming forum, and via word-of-mouth. Participants needed to have a solid background in programming, with enough experience in the Java programming language to read Java code and a basic understanding of multithreaded code. We recruited 53 participants, with 4–30 years of programming experience. Participants included undergraduates in a computer science or related degree program, graduate students in a computer science or related degree program, and industry professionals. Only 32 participants made it beyond the preliminary questions to the multithreaded code samples. Of these 32 participants, 4 were disqualified due to leaving all the bug questions blank, not finishing the survey, and/or having multiple errors in documentation and multithreading concepts questions.

Participants were separated into two categories via self selection. *Novice* users report they can count on one hand the number of multithreaded programs they have written. They are cognizant of some of the difficulties present in writing code with concurrency (*e.g.*, race conditions), but not very familiar with multithreading practices or techniques. *Expert* users report they have frequent experience with multithreaded code. Unfortunately, the distribution of novice and expert users was not uniform across the two groups. More novice users ended up in the group with annotations (Figure 3). This may have added a bias against the annotation group.

	Novice	Expert	Total
Yield Annotations	7	6	13
No Yields	4	11	15
Total	11	17	28

Figure 3. Distribution of participant experience.

4.3 Pilot

We ran a pilot study with 5 participants and some additional code samples. The pilot was a within-group study: we presented participants with 5 code samples twice, first without and then with yield annotations. All participants found the survey to be **much** too long, so we switched to a between-group study design and reduced the number of code samples. We also clarified the instructions, and emphasized that the survey was not a test. All pilot participants agreed that yield annotations were helpful for them to understand the code. In the case of the `StringBuffer` example (which was used in both the pilot and final study), only one participant was able to correctly identify the concurrency bug in the annotationless code, but all participants could find the bug in the code with annotations.

5. Results

We coded the qualitative responses of participants identifying code bugs in each sample on an A, B, C scale. An A response is one that correctly identifies the seeded concurrency bug. Confused and/or incorrect responses were rated as C responses; these include identification of non-existent bugs or erroneous statements about the code samples. B responses identify valid issues in the code samples (*e.g.* stylistic issues or potential sequential bugs), but do not identify the seeded concurrency bug. Because our sample size was small, we used the Fisher-Irwin test to check for statistical

significance [1]. We deem a result with a p-value below 0.05 to be statistically significant.

ArchivalList Although a higher proportion of participants in the annotation case correctly identified the multithreaded code bug (Figure 4), this result is *not* statistically significant.

	A	B	C	Total
Yield Annotations	10	1	2	13
No Yields	8	1	6	15
Total	18	2	8	28

Figure 4. Quality of bug reports for `ArchivalList`.

Despite this, a statistically significant higher proportion of participants in the annotation case had confidence with the statement "I have identified any bugs in the code" (p-value $= .046$).

IntList Three people skipped the questions about this code sample. Of the remaining 25 participants, we found that having yield annotations correlates (p-value $= .020$) with correctly identifying multithreaded bugs (Figure 5).

	A	B	C	Total
Yield Annotations	10	1	0	11
No Yields	8	0	6	14
Total	18	1	6	25

Figure 5. Quality of bug reports for `IntList`.

The two skipping participants in the with-annotation group received A and B scores for the other code samples. Conversely, the participant in the without-annotation group scored a C and a B on the other code samples. If all three skipping participants had received a B for this code sample, the correlation would still be statistically significant.

StringBuffer One participant skipped this question. Of the remaining 27 participants, we found that having yield annotations highly correlates (p-value $< .001$) with correctly identifying multithreaded bugs (Figure 6). The correlation would still be statistically significant even if this participant had received a C score for this code sample.

	A	B	C	Total
Yield Annotations	10	1	1	12
No Yields	1	5	9	15
Total	11	6	10	27

Figure 6. Quality of bug reports for `StringBuffer`.

5.1 Feedback

Most participants (10 out of the 12 that answered this question) who had yield annotations in their code either agreed or strongly agreed that yield annotations helped them reason about multithreaded code. Participants found that annotations helped them find tricky points in the code and revisit incorrect assumptions:

> *"Draws attention to region that could/does cause problems. Helps direct thought process about what can go wrong and what else needs to be done."*

> *"I hadn't been thinking about how the synchronized command didn't protect the contents of passed in parameters from being modified."*

The first dissenting participant admitted that the annotations were actually helpful, but was concerned about the potential for over-reliance on yields leading to overconfidence about code correctness:

> *"To some extent, they helped me find issues with the existing code. . . . These annotations don't help you find what resources are shared and could lead to younger programmers making dangerous assumptions about atomicity."*

The second dissenting participant did not understand the annotations:

> *"I felt the yield annotations drew my focus away from synchronized blocks."*

We realized this study was also implicitly about how well we explain yield annotations, and how intuitive they are. In future studies, more care needs to be taken when providing background material, such as the semantics of yield annotations.

6. Discussion

6.1 Hypotheses

As a first study to evaluate cooperability, the results are encouraging but not conclusive. Most participants find yield annotations to be helpful.

H1 Our results support this hypothesis. In two out of three code samples, yield annotations are associated with a statistically significant increase in identification of concurrency bugs.

H2 Our results are inconclusive for this hypothesis. For the the `ArchivalList` example, yield annotations are associated with a statistically significant increase in confidence of bug identification. However, in two out of three code samples, yield annotations are not correlated with stronger impressions of confidence.

H3 Our results do not support this hypothesis. We found *no* statistically significant difference in the quality of bug reports found between experts vs. novices. We would like to revisit the question of what it means to be an "expert" at multithreaded programming.

6.2 Lessons Learned

The study caused us to revisit our own assumptions about how to explain cooperability. By running this user study, we were able to test how well cooperability was conveyed to experienced programmers unfamiliar with the theoretical back-

ground. We advocate user studies for "intuitive" concepts as a way to discover hidden assumptions and biases.

Although understanding the learning curve for a language or system is important, studies that compare development time for novice users are expensive, and often inconclusive (see the Related Work section of this paper). We advocate lightweight surveys which go beyond self-assessment to also involve problem solving questions, perhaps combined with in-depth interviews with experienced developers.

6.3 Threats to Validity

Threats to validity for this study include the relatively small sample size, the potential non-representativeness of the participants, and the effect of participants who skipped questions or did not complete the survey. Also, the results could be biased from the particular code samples we chose, the fact that they were small enough for participants to digest quickly, and the small number of code samples.

7. Future Work

This study is an initial evaluation of cooperability; we have three main future directions to explore. We would like to evaluate the use of yield annotations in the code development process by conducting interviews with programmers in which they modify code containing yield annotations. We are also exploring the question of how yield annotations can be made more usable. We are currently exploring the use of discount usability methods [11] such as heuristic evaluation to identify improvements. Lastly, we are characterizing concurrency bugs for which yield annotations are most useful.

Additional questions exist within the larger research area of parallel programming and usability. We would like to investigate how users debug and modify parallel programs, and how effective existing parallel programming systems are in this process. We want to better understand the correlation between actual ability and perceptions of ability to solve concurrency problems. We would also like to apply HCI techniques to study how to teach programmers about correctness conditions for parallel code. For example, the importance of effective metaphors is often overlooked, but can have a significant impact on understanding, particularly for novice users of a system [12].

Acknowledgments

Special thanks to Sri Kurniawan, Alexandra Holloway, Ian Pye, and Cormac Flanagan for experiment feedback and participant recruitment. Thanks to all the pilot testers, who had to undergo 1.5 hours of survey, and to all the people that took the (shorter) refined version. Thanks to the reviewers for their constructive comments.

References

[1] I. Campbell. Chi-squared and Fisher-Irwin tests of two-by-two tables with small sample recommendations. *Statistics in Medicine*, 26:3661–3675, 2007.

[2] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, J. Urbanic, and P. Center. An experiment in measuring the productivity of three parallel programming languages. In *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2006.

[3] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages (POPL)*, 2004.

[4] C. Halverson and J. Carver. Climbing the plateau: Getting from study design to data that means something. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2009.

[5] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *Conference on High Performance Networking and Computing (SC)*, 2005.

[6] L. Hochstein, V. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81(11): 1920–1930, 2008.

[7] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[8] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Notices*, 2008.

[9] M. Luff. Empirically investigating parallel programming paradigms: A null result. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2009.

[10] B. Myers and A. Ko. The past, present and future of programming in HCI. In *Human-Computer Interaction Consortium (HCIC)*, 2009.

[11] J. Nielsen. *Usability Inspection Methods*. Wiley, 1994.

[12] J. Pane and B. Myers. Usability issues in the design of novice programming systems. *Human-Computer Interaction Institute Technical Report CMU-HCII-96-101*, 1996.

[13] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[14] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[15] C. Rossbach, O. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[16] D. Szafron, J. Schaeffer, and A. Edmonton. An experiment to measure the usability of parallel programming systems. *Concurrency Practice and Experience*, 8(2):147–166, 1996.

[17] J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2010.