

# Using Formal Models to Objectively Judge Quality of Multi-Threaded Programs in Empirical Studies

Laura K. Dillon<sup>†</sup>, R. E. K. Stirewalt<sup>†</sup>, Eileen Kraemer<sup>†‡</sup>, Shaohua Xie<sup>‡</sup>, Scott D. Fleming<sup>†</sup>  
<sup>†</sup>Dept. of Computer Science and Engineering  
Michigan State University  
East Lansing, Michigan, USA 48824  
{ldillon, stire, kraemere, sdf}@cse.msu.edu

<sup>‡</sup>Department of Computer Science  
University of Georgia  
Athens, Georgia, USA 30602-7404  
{eileen, shaohua}@cs.uga.edu

## Abstract

*Empirical studies are important for understanding how well current design methods and notations support development of multi-threaded programs. Unfortunately, concurrency exacerbates an already difficult problem in drawing conclusions from such studies: How to objectively measure the quality of candidate solutions produced by participants in the studies. This paper explores the use of formal modeling and analysis for this purpose. We describe initial findings of a small pilot study to determine if we can objectively differentiate sample candidate solutions with respect to their use of synchronization primitives. To do so, we faithfully model these candidate solutions and various synchronization-related properties in the Finite State Processes (FSP) notation and use the Labeled Transition System Analyzer (LTSA) to analyze the solution models against the properties.*

## Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent programming;  
D.2.4 [Software Engineering]: Software/Program Verification—  
formal methods, model checking

## General Terms

Experimentation, Verification

## Keywords

Finite State Processes (FSP), Labeled Transition System Ana-

lyzer (LTSA), quality assessment of code, synchronization analysis

## 1. Introduction

Concurrent software designs are increasingly prevalent and yet very difficult to design and verify. We have been conducting empirical studies to assess how well various modeling methods and notations assist programmers in these activities [11, 4]. During these studies, participants are broken into two groups—a treatment group that uses the modeling notation or method under study and a control group that does not—and they are asked to develop, extend, or correct the source code of a multi-threaded program. We then judge these solutions according to a battery of quality measures (e.g., freedom from deadlock, frequency of unnecessary signals, etc.) and look for statistically significant differences in quality between the two groups. Often, we must judge a single artifact with respect to a large number of quality measures, and the soundness of any conclusions we draw relies on the objectivity of these judgments. This paper explores the idea of formalizing quality measures in terms of safety properties and verifying them against *faithful models* of the synchronization structure of the submitted source code.

A synchronization model is faithful to the code if (1) it is structured so that elements in the model correspond directly to code statements and structures and vice versa, and (2) it explicitly models low-level synchronization primitives (e.g., mutex locks and condition variables) and operations on shared data. By virtue of these properties, models are trivially traceable to code. Thus, synchronization flaws in the code will be preserved in the model rather than abstracted away, as may easily happen when constructing an *ad hoc* model of a program with a subtle flaw. This concern is especially relevant given the large number of such models that might need to be constructed.

Others have encountered the need to judge source code according to multiple quality measures in the context of an empirical study [3, 5]. Ideally, a distinct test case can be developed to assess each measure in isolation. However, code solutions produced during an empirical study may fail to compile or may take the form of code snippets rather than complete programs. Moreover, many non-functional qualities (e.g., extensibility) are not testable at all, and others are not reliably testable. Qualities related to correct use of synchronization primitives are good examples of the lat-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MiSE'08, May 10–11, 2008, Leipzig, Germany.

Copyright 2008 ACM 978-1-60558-025-8/08/05 ...\$5.00.

ter. Some researchers resort to subjective measures in these cases (e.g., [1, 8]); however, such measures may threaten validity and are difficult to weight relative to one another and to more objective measures.

As a first step to explore our ideas for judging the quality of synchronization code, we conducted a pilot study designed to answer two questions:

1. Given code that uses locks and condition synchronization to implement a shared resource, can we systematically generate a model that exhibits the same synchronization behavior as the code and that is compact enough to be feasibly analyzed?
2. Can we draw conclusions about the relative quality of the code through automated analysis of such models?

For the pilot study, we used candidate programs written by participants in a larger empirical study (Section 2). From a library of reusable models of synchronization primitives, we created faithful models of 5 representative candidate solutions (Section 3). We then subjected these models to a battery of automated analyses intended to assess different measures of solution quality (Section 4). This paper presents the results of the pilot study (Section 5). A rich variety of contexts and properties were found to provide an objective means for judging the quality of different solutions with respect to different quality measures (Section 6). Finally, we offer some concluding remarks (Section 7).

## 2. Problem Description

Our study asked participants to write C++ code for a variant of the party-rendezvous problem that we call the *match maker* problem. For this problem one or more matchmaker threads attempt to pair up girls and boys who have arrived at a party. Figure 1 depicts an elided solution to this problem. Class `MatchMaker` declares a mutex `lock`, two condition variables `goGirls` and `goBoys`, and three counter variables, `nGirls`, `nBoys`, and `nPairs`. Although not shown in this figure, the condition variables are both initialized with a reference to `lock`; thus, calls to wait on these entities will cause `lock` to be released. *Clients* running in different threads synchronize with one another using this class as follows: to send a boy or a girl to the matchmaking service, a client invokes the operation `addBoy` or `addGirl`, respectively. To pair up a girl and a boy that were previously sent to the matchmaking service, a client invokes the operation `pair`. The `addBoy` and `addGirl` operations never cause a client to wait. However, if a client invokes `pair` when all the girls or all the boys that were previously sent to the matchmaking service have already been paired, the client must wait until both an unpaired boy and an unpaired girl are available.

Our study provided every participant an English description of the matchmaker problem and a version of Figure 1 with only prototype declarations of the member functions `addBoy`, `addGirl` and `pair`. We then split the participants into two groups. Those in the treatment group were first asked to draw a UML 2.0 state diagram depicting the intended behavior of the matchmaker and then later asked to fill in the method bodies for the three operations, while those in the control group were first asked to fill in the method bodies and then later asked to draw a UML 2.0 state diagram. In the end, only a handful of candidate solutions produced by the 52 participants in this study looked like the code in

```
class MatchMaker {
    ACE_Thread_Mutex lock;
    ACE_Condition_Thread_Mutex goGirls, goBoys;
    unsigned nGirls, nBoys, nPairs;
    ...
    void pair() {
        lock.acquire();
        while (nBoys<=nPairs || nGirls<=nPairs) {
            if (nBoys<=nPairs)
                goBoys.wait();
            if (nGirls<=nPairs)
                goGirls.wait();
        }
        nPairs++;
        lock.release();
    }
    void addBoy() {
        lock.acquire();
        nBoys++;
        goBoys.signal();
        lock.release();
    }
    void addGirl() {
        lock.acquire();
        nGirls++;
        goGirls.signal();
        lock.release();
    }
};
```

Figure 1. Class `MatchMaker`.

Figure 1, and no solution was without synchronization errors. To draw any valid conclusions from this study, therefore, we were faced with the problem of comparing the candidate solutions for qualities other than strict correctness. The remaining sections report on a pilot study to determine if we could feasibly do so by faithfully modeling each candidate solution as a process in the Finite State Processes (FSP) notation and analyzing the processes so obtained for various properties.

## 3. Faithful Models

Our models of candidate solutions are distinguished by two features. First, they model OS-level resources, such as mutex locks, semaphores, and condition variables, as distinct behavioral entities. Our models borrow heavily on the conventions of Magee and Kramer [10, ch. 13], especially their model of condition variables. Second, model components are defined to correspond one-to-one to synchronization-relevant statements in the program and to compose according to composition of statements in the program. FSP models are nice in this regard: Elementary program statements are modeled as *sequential processes* and composed using sequential composition, thereby mimicking the composition of statements in the program, and concurrency is modeled using parallel composition. These concepts are best illustrated by example.

Figure 2 depicts the definition of an FSP process that models an instance of class `MatchMaker` as a *shared resource* using the idiom described in [10, sect. 3.1.3]. Under this idiom, operations in a shared resource manifest as actions, which are labeled by the set of threads that invoke the operation. `MATCH_MAKER_SHARED` is a composite process whose components correspond one-to-one

```

| MATCH_MAKER_SHARED =
  ( LOCK_THREADS :: MATCH_MAKER_LOCK | |
    MATCH_MAKER_GO_GIRLS_CVAR | |
    MATCH_MAKER_GO_BOYS_CVAR | |
    NUM_GIRLS_THREADS :: MATCH_MAKER_NUM_GIRLS | |
    NUM_BOYS_THREADS :: MATCH_MAKER_NUM_BOYS | |
    NUM_PAIRS_THREADS :: MATCH_MAKER_NUM_PAIRS ) . . .

```

**Figure 2. MatchMaker as a shared resource.**

with the data members of class `MatchMaker`, as depicted in **Figure 1**. The primitive process components (e.g., `MATCH_MAKER_LOCK`, `MATCH_MAKER_NUM_GIRLS`, etc.) are simply instances of reusable and pre-defined MUTEX and COUNTER processes, which are then labeled with the name of this resource (not shown). The `xxx_CVAR` processes model the condition variables `goGirls` and `goBoys`. These composite processes are instantiated in another part of the model from predefined processes using the idiom from [10, ch. 13].

By convention, the sets labeled `xxx_THREADS`, called *accessor thread sets*, contain the names of threads that access resource (or invoke operation) `xxx`. For instance, `LOCK_THREADS` names those threads that acquire or release the mutex lock; whereas `NUM_GIRLS_THREADS` names those threads that increment, decrement, or read the value of the counter variable `nGirls`. We use accessor thread sets so as to abstract the names of the actual threads out of the definition of the shared resource. This simplifies the assembly of analysis models, which will employ different configurations of threads. We also use accessor thread sets to instantiate the condition variables (`goGirls` and `goBoys`). This instantiation (not shown for lack of space) is more involved, but is nonetheless mechanical. We also elide a renaming of actions (ellipsis in the figure) needed to guarantee correct synchronization with client threads.

To model the operations of class `MatchMaker`, we first model each statement that directly accesses or modifies a synchronization object or counter variable as a sequential process<sup>1</sup> in FSP. We then use FSP’s sequential composition and branching primitives to compose these primitive processes according to the control flow graph of a participant’s submitted code.

**Figure 3** depicts some of the FSP processes we defined to model the `pair` method from **Figure 1**. The first three processes model statements that acquire the lock, release the lock, and increment `nPairs`, respectively. Process `PAIR_OP` is the sequential composition of subprocesses, each of which models one of the top-level statements of the method. The most interesting subprocess, `PAIR_OP_LOOP`, models the while loop. It models checking the while condition using actions that read the counters `nBoys` and `nPairs`. Based on the values read, it calculates the value of the first disjunct of the while condition. If true, control transfers into the body of the loop and then the process repeats. Otherwise, the second disjunct is checked (not shown in figure). Process `WHILE_BODY` (elided for brevity) is the sequential composition of two smaller processes—`FIRST_IF` (shown) and `SECOND_IF` (not shown). Process `FIRST_IF` models the first if statement inside the body of the loop. Process `WAIT_GO_BOYS` models invocation of the wait statement inside this first if block according to the idiom in [10, ch. 13].

<sup>1</sup>i.e., a terminating process—one that ultimately evolves into the primitive process `END`.

```

ACQUIRE_LOCK = (lock.acquire -> END).
RELEASE_LOCK = (lock.release -> END).
INC_PAIRS = (nPairs.inc -> END).

PAIR_OP = ACQUIRE_LOCK;
          PAIR_OP_LOOP;
          INC_PAIRS;
          RELEASE_LOCK;
          END.

PAIR_OP_LOOP =
  (nBoys.read[nb:BRANGE] ->
   nPairs.read[np:PRANGE] ->
   if (nb <= np)
   then
     WHILE_BODY;
     PAIR_OP_LOOP
   else...)

FIRST_IF =
  (nBoys.read[nb:BRANGE] ->
   nPairs.read[np:PRANGE] ->
   if (nb <= np)
   then WAIT_GO_BOYS; END
   else END).

WAIT_GO_BOYS =
  (goBoys.wait ->
   lock.release ->
   goBoys.endwait ->
   lock.acquire -> END).

```

**Figure 3. Faithful model of the pair method.**

These process definitions model the code to a high degree of fidelity. Process `PAIR_OP_LOOP` models reading the values of the counter variables in the order these reads would actually occur. Also, the process is structured to short-circuit evaluation of the second disjunct if the first disjunct is true. These process definitions could be automatically assembled from a sufficiently rich collection of primitive processes. Fortunately, in an empirical study, we usually know the primitive resources available and all operations *a priori*. In practice, these are often given as part of the problem statement.

## 4. Analysis Models and Properties

Prior to analysis, the FSP process created to model a shared resource must be composed with a process that models clients that actively invoke operations on the resource. We refer to the process modeling the shared resource as the *resource model*, the process modeling the clients as the *client model*, and to the parallel composition of the resource model and the client model as the *analysis model*. The Labeled Transition System Analyzer (LTSA) is used to check an analysis model for properties which may reveal the presence (or demonstrate the absence) of specific synchronization errors.

Of course, the properties satisfied by an analysis model depend not just on the resource model, but also on the client model. This fact is important, especially when most of the candidate solutions have synchronization errors, as it permits us to differentiate resource models that exhibit the same synchronization error. Consider, for example, the 5 client models described in **Figure 4** (bottom left and right), which are easily expressed in FSP

as processes. These client models form an ordered set of progressively more concurrent (less restrictive) models, ranging from a strictly sequential client model (SRou) to a fully concurrent one (FCon). The more restrictive client models tend to mask certain concurrency errors. Moreover, if the parallel composition a given resource model with a given client model does not satisfy a given safety property, then the parallel composition of the given resource model with any less restrictive client model also does not satisfy the given safety property. Thus, for each safety property that expresses a desired quality of a solution, the ordering of the 5 client models induces an ordering on the 5 analysis models obtained using a given resource model. The analysis model in which a synchronization error first manifests provides a measure of the quality of the resource model (and thus of the associated candidate solution). The results of our pilot study illustrate this phenomenon more concretely (Section 5).

LTSA automatically checks the analysis models for deadlocks. To assess other quality measures with LTSA, we defined 3 problem-specific properties. The first checks that a resource model conforms to the standard protocol for locking a shared resource—i.e., that a thread holds the lock when it invokes an operation on the data members of the shared resource and that it no longer holds the lock when it returns from an invocation of `addBoy`, `addGirl`, or `pair`. The second property checks that boys and girls are paired correctly—i.e., it tracks the number of times each counter has been incremented and checks that the number of increments of `nPairs` never exceeds the number of increments of `nBoys` or of `nGirls`. The third process checks for excessive signaling. While not a correctness requirement, this latter check provides insight into quality of a resource model. The definition of excessive signaling is both problem- and client model-specific. For our problem and a client model that calls each method 3 times, the analysis model should generate a maximum of 6 signals (one for each invocation of `addBoy` or `addGirl`).

## 5. Preliminary Findings

Figure 4 summarizes the results from our pilot study. We created resource models from the correct solution depicted in Figure 1 (indicated as `Correct`) and from 5 representative candidate solutions (indicated using participant numbers) submitted by participants the larger empirical study. We also created 5 client models (`SRou`, `CRou`, `TPMe`, `CPai`, and `FCon`). These represent different configurations of threads executing in client code, as described in the tables titled `Client Models` (bottom left and right). The 6 resource models and 5 client models were composed, producing 30 analysis models. Each analysis model was checked for deadlock (top left), conformance to the lock protocol (top right), correct pairing of boys and girls (middle left), and excessive signaling (middle right). To show the results of safety checks, we mark: “0,” if the model satisfies the property; “X,” if the model may violate the property; and “–,” if analysis is inconclusive.

We consider analyses inconclusive for two reasons, only one of which we anticipated. An analysis model composed from a faithful resource model can easily be too large for exhaustive analysis to be feasible. However, we expected that the solutions submitted by participants in our empirical study were simple enough that faithful models would also be analyzable, and indeed, for the most part, this was the case. We had to reduce the number of times each

method is invoked from 3 to 2 in the last 2 client models (`CPai` and `FCon`) in order not to exceed the Java stack size when using LTSA with these models. Fortunately, the properties checked in the pilot study did not depend on the precise number of method invocations so long as each method was invoked at least twice and the same number of times.

The second reason that analysis might be inconclusive was not anticipated going into the study. When checking safety of a composite FSP process, LTSA reports that the composite process deadlocks if some trace leads to either `ERROR` or `STOP`,<sup>2</sup> and it returns a shortest such trace. However, when the composite FSP process is the parallel composition of an analysis model and a safety property (as is the case for the tables `Lock Protocol`, `Correct Pairing` and `Excessive Signaling`), we can conclude that the analysis model does not satisfy the safety property only if the composite model can reach `ERROR`. Thus, in cases where LTSA returns a trace that leads to `STOP` rather than `ERROR`, we cannot conclude that the property does not hold. In fact, none of the candidate solutions used in this study violated the standard locking protocol; however, as shown in the table for `Lock Protocol`, we could not verify this fact for 12 of the analysis models (the 12 that could deadlock). Note that this problem is an artifact of how LTSA checks safety properties, and is not intrinsic to model checking. This source of inconclusive results can be eliminated by adding an option designating a search for traces that lead to `ERROR` only.

## 6. Discussion

One question we strove to answer was whether the quality of candidate solutions can be judged using automated analysis of faithful models. In our study, we used these analyses to make a number of useful judgements. For instance, only participant #3’s submission can deadlock in a context in which a pair operation should never have to wait (client model `CRou`). Moreover, the submissions of the other participants pair the boys and girls correctly in this context. These observations provide an objective basis for ranking the quality of the submission of participant #3 lower than that of the others, at least with regard to preventing deadlock, and possibly also with regard to correct pairing. Similarly, the submissions of participants #3 and #4 rank lower with regard to these same 2 properties than those submitted by #1 and #2 in the context where activations of different operations execute concurrently but activations of the same operation execute sequentially (client model `TPMe`). Regarding deadlock, the submission of participant #1 ranks below that of participant #2, as the former can deadlock in a context (represented by `CPai`) that may invoke pair operations concurrently. Regarding correct pairing, however, these latter two submission rank the same.

The results of checking conformance to the locking protocol provide no basis for ranking any candidate solution better than than any other. This is as it should be because every submission correctly conformed to this protocol. Regarding excessive signaling, the results indicate the submission of participant #2 ranks higher than that of participant #3, which in turn ranks higher than those of #1 and #4, and that these latter 2 submissions rank the same. We do not include the submission of participant #5 in this

<sup>2</sup>primitive FSP processes which have no successors, but serve different purposes: `STOP` represents a deadlocked process, whereas `ERROR` serves as a trap state when checking properties.

Deadlock					
Participant	SRou	CRou	TPMe	CPai	FCon
Correct	0	0	0	0	0
#1	0	0	0	X	X
#2	0	0	0	0	0
#3	0	X	X	X	X
#4	0	0	X	X	X
#5	0	0	X	X	X

  

Correct pairing					
Participant	SRou	CRou	TPMe	CPai	FCon
Correct	0	0	0	0	0
#1	0	0	0	X	X
#2	0	0	0	X	X
#3	0	-	X	X	X
#4	0	0	X	X	X
#5	0	0	-	-	-

  

Lock protocol					
Participant	SRou	CRou	TPMe	CPai	FCon
Correct	0	0	0	0	0
#1	0	0	0	-	-
#2	0	0	0	0	0
#3	0	-	-	-	-
#4	0	0	-	-	-
#5	0	0	-	-	-

  

Excessive signaling					
Participant	SRou	CRou	TPMe	CPai	FCon
Correct	0	0	0	0	0
#1	X	X	X	X	X
#2	0	0	0	0	0
#3	0	X	X	-	-
#4	X	X	X	X	X
#5	0	0	-	-	-

  

Client Models	
SRou	(Sequential Rounds) 1 thread calls <code>addBoy</code> , <code>addGirl</code> , & <code>pair</code> in sequence, 3 times
CRou	(Concurrent Rounds) 3 threads, each calls <code>addBoy</code> , <code>addGirl</code> , & <code>pair</code> in sequence, once
TPMe	(Thread Per Method) 1 thread per method, each calls its method 3 times

  

Client Models	
CPai	(Concurrent Pairs) 2 threads call <code>pair</code> once, 1 thread calls <code>addBoy</code> twice, & 1 thread calls <code>addGirl</code> twice
FCon	(Fully Concurrent) 2 threads call <code>addBoy</code> once, 2 threads call <code>addGirl</code> once, & 2 threads call <code>pair</code> once

**Figure 4. Results of analyzing resource models corresponding to the correct solution (“Correct”) and the solutions produced by 5 participants (“#1”–“#5”) for deadlock (top left table), adherence to the locking protocol (top right table), correct pairing of boys and girls (middle left table), and excessive signalling (middle right table), and using 5 client models (bottom left and right tables). Results are indicated as: “0,” if the property is satisfied; “X,” if the property is not satisfied; and “-,” if analysis is inconclusive.**

ranking because of the inconclusive results. In contrast, because the submission of participant #3 can exhibit excessive signaling in the context represented by `CRou`, it can also do so in the less restrictive contexts (`CPai` and `FCon`). Thus, we include this participant in the ranking, in spite of the dashed entries.

Another question we strove to answer was whether the generation of analysis models and properties is sufficiently systematic. We produced the client models and resource models manually, and then automatically assembled them into analysis models using the Noweb literate programming tool [7]. The resource models were systematically assembled from a library of reusable models of synchronization primitives and from simple models of program statements. For the candidate solutions produced by participants in our larger empirical study, this assembly process is certainly automatable. Additionally, given specifications of the numbers of threads and of the operations each thread performs, the client models could be automatically generated. To generate the analysis models for the pilot study, we manually created a Noweb document containing the resource models, the client models, and specifications for assembling them into analysis models. The assembly specifications were produced by instantiating a single template. Thus, we were able to systematically produce faithful FSP models and it would be feasible to automatically generate them for a larger follow-up study that uses the same problem.

That said, generating faithful FSP models of candidate solutions to general concurrent programming problems is not fully automatic. Substantial effort is required to create the problem-specific infrastructure needed to systematically generate the analysis models and properties. Regarding the properties: No additional work is required to check for deadlock, but all the other checks re-

quire a property describing legal behaviors. The properties are reusable across all resource models created from the candidate solutions to the `MatchMaker` problem. This is to be expected, as a problem typically defines the properties that its solutions must satisfy.

We used FSP/LTSA for the pilot study because we felt FSP models could be both faithful to the code and suitably abstract. Faithfulness is important for ensuring the quality of the analysis model accurately reflects the quality of the code. Abstractness is necessary in order that exhaustive analysis is feasible. But abstractness can easily compromise faithfulness, so a balance is needed. Using tools, such as Java PathFinder [6] or Bandera [2], that extract models directly from source code, might save on effort needed to develop problem-specific infrastructure. We could not use these tools for our pilot study because our participants produced C++ code using synchronization primitives supplied by a library. In summary, we felt that FSP provided an appropriate balance.

## 7. Concluding Remarks

No statistically meaningful conclusions can be drawn from our pilot study because of its small size. Our purpose was to relatively quickly assess the likelihood that we might obtain some interesting and statistically meaningful results by subjecting the full suite of candidate solutions to such analysis and whether a study of this size would be feasible to perform. We are encouraged by the results. Although no participants in the larger empirical study submitted correct solutions, our battery of analyses should permit us to rank them with respect to at least the measures described in **Sec-**

tion 4, as well as to 2 additional measures which we found useful in the pilot study (but omitted in the paper for lack of space). Thus, we might find significant differences between participants from the different treatment groups on one or more quality measures.

For a larger study, the generation of the analysis models will have to be automated. Moreover, we will need to check properties of the models in batch mode and the results of the checks will need to be output in a processable form. Although we did not see how to run batch analyses or how to output traces or runs for subsequent processing with the version of LTSA available at [9], it should be possible to obtain a version of the tool with these capabilities. Having already created the infrastructure for the pilot study, expanding the pilot study should now be straightforward.

**Acknowledgements:** Partial support was provided for this research by NSF grants CCF 0702667, IIP 0700329 and IIS 0308063, and by LogicBlox Inc.

## 8. References

- [1] E. Arisholm et al. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.
- [2] M. B. Dwyer et al. Tool-supported program abstraction for finite-state verification. In *Proceedings of the International Conference on Software Engineering*, pages 177–187, 2001.
- [3] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, 2005.
- [4] S. D. Fleming, E. T. Kraemer, R. E. K. Stirewalt, S. Xie, and L. K. Dillon. A study of student strategies for the corrective maintenance of concurrent software. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE'08)*, 2008.
- [5] B. George and L. Williams. An initial investigation of test driven development in industry. In *SAC '03: Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 1135–1139, New York, NY, USA, 2003. ACM.
- [6] K. Havelund and T. Presburger. Model checking Java programs using Java Pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 1998.
- [7] A. Johnson and B. Johnson. Literate programming using Noweb. *Linux J.*, page 1, Oct. 1997.
- [8] A. Karahasanović and R. C. Thomas. Difficulties experienced by students in maintaining object-oriented systems: an empirical study. In *Proceedings of the 9<sup>th</sup> Australasian Conference on Computing Education*, pages 81–87, 2007.
- [9] Download LTSA V3.0.  
<http://www.doc.ic.ac.uk/~jnm/book/ltsa/download.html>.
- [10] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, second edition, 2006.
- [11] S. Xie, E. T. Kraemer, and R. E. K. Stirewalt. Empirical evaluation of a UML sequence diagram with adornments to support understanding of thread interactions. In *Proceedings of the IEEE International Conference on Programming Comprehension*, June 2007.