Reducing the Barriers to Writing Verified Specifications

Todd W. Schiller Michael D. Ernst
University of Washington
{tws,mernst}@cs.washington.edu

Abstract

Formally verifying a program requires significant skill not only because of complex interactions between program subcomponents, but also because of deficiencies in current verification interfaces. These skill barriers make verification economically unattractive by preventing the use of less-skilled (less-expensive) workers and distributed workflows (i.e., crowdsourcing).

This paper presents VeriWeb, a web-based IDE for verification that decomposes the task of writing verifiable specifications into manageable subproblems. To overcome the information loss caused by task decomposition, and to reduce the skill required to verify a program, VeriWeb incorporates several innovative user interface features: drag and drop condition construction, concrete counterexamples, and specification inlining.

To evaluate VeriWeb, we performed three experiments. First, we show that VeriWeb lowers the time and monetary cost of verification by performing a comparative study of VeriWeb and a traditional tool using 14 paid subjects contracted hourly from Exhedra Solution's vWorker online marketplace. Second, we demonstrate the dearth and insufficiency of current ad-hoc labor marketplaces for verification by recruiting workers from Amazon's Mechanical Turk to perform verification with VeriWeb. Finally, we characterize the minimal communication overhead incurred when VeriWeb is used collaboratively by observing two pairs of developers each use the tool simultaneously to verify a single program.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords program verification, human factors, crowdsourcing

1. Introduction

Ideally, in a well-modularized program, it would be possible to perform software engineering tasks on each module locally. Nonetheless, some software engineering tasks require, or benefit from, a global view. For example, it is difficult to decompose the task of writing method contracts (formal specifications) into independent subproblems because a change to one method contract may require changes to others as a result of interdependencies between program elements — a method's contract may depend on the contracts for other methods it calls. Likewise, the task of writing an application or object specification would likely benefit from considering multiple object and methods specifications simultaneously. Attempting to decompose tasks like these results in information (context) loss, the severity of which depends on the quality of the program's design and documentation; incomplete information leads to confusion, mistakes, and wasted or duplicated effort. For writing and verifying formal specifications, the difficulty is exacerbated by the complexity and inaccessibility of current verification interfaces. Coupled with a high cost of labor (the average developer in the United States earns $90,170 per year [35]), the result is that the time and money costs of verification outweigh the benefits for the general software development community.

Tools and techniques that enable less-skilled workers to perform verification would make formally establishing correctness more economically attractive. To the extent that such tools decompose the problem, the tools would further improve the economics of verification by enabling parallel and distributed workflows (e.g., crowdsourcing to workers in a global marketplace).

VeriWeb Verification Interface This paper presents VeriWeb, a tool for creating verifiable Java Modeling Language (JML) [6, 22] specifications for existing programs by leveraging the "crowd," a (potentially distributed) set of workers solving problems via web client. Figure 1 shows a partial example of a method contract, expressed in JML.

VeriWeb decomposes the larger task of writing a verifiable program specification into smaller subtasks: creating one method's preconditions and postconditions. To compensate for the effects of decomposition and to lower the tool's skill requirement, VeriWeb includes several novel interface

```
class Queue {
  int /*@spec_public*/ currentSize;
  /*@invariant currentSize >= 0; */
  ...

  /*@ requires x != null;
    @ ensures currentSize == \old(currentSize) + 1;
    @ exsures (RuntimeException) ... */
  public void enqueue(Object x)
    throws RuntimeException {
    ...
  }
}
```

Figure 1. An example partial Java Modeling Language (JML) contract for the enqueue method of a queue. "Requires" clauses state properties that must hold when the method is called. "Ensures" clauses state properties that must hold when the method exits normally. "Exsures" clauses state properties that must hold when the method throws the declared exception. "Invariants" state properties that must hold whenever an object is visible, e.g., after a constructor or a public method call. A tool such as ESC/-Java2 [9] can verify that the program meets the specification, and that no unexpected exceptions (e.g., NullPointerException) will be thrown.

features: drag and drop contract construction, concrete counterexamples, contract inlining, and context clues. VeriWeb additionally includes contract suggestions inferred from dynamic traces. VeriWeb can be viewed as an IDE for verification, where the combined effect of the features is greater than the sum of the individual parts.

By enabling the use of less-skilled labor for verifying an existing program, VeriWeb admits the following workflow for skilled feature developers:

1. The skilled developer writes a program or feature.
2. The skilled developer (optionally) writes a partial JML specification of the program or feature.
3. The skilled developer submits the program and partial specification to VeriWeb, which utilizes the "crowd" to complete a verifiable JML specification.

Extended Static Checking Internally, VeriWeb uses ESC/-Java2 [9] to perform Extended Static Checking [4, 7, 13, 14, 19, 24] of the program against the generated specification. Extended Static Checking is a verification approach that can be viewed as a front-end or user interface to an automated theorem prover [11, 12, 27]. The user writes a specification consisting of method preconditions, method postconditions, and object invariants. (Figure 1 shows a partial example of a method contract, expressed in the Java Modeling Language (JML) [6, 22].) The extended static checker converts both the specification and the program code into logical formulas, passes the combined formulas to the theorem prover, and reports to the user whether or not the program satisfies the specification. If not (that is, the verification attempt failed), the user revises the specifications or the code using the feedback from the checker, and then tries again.

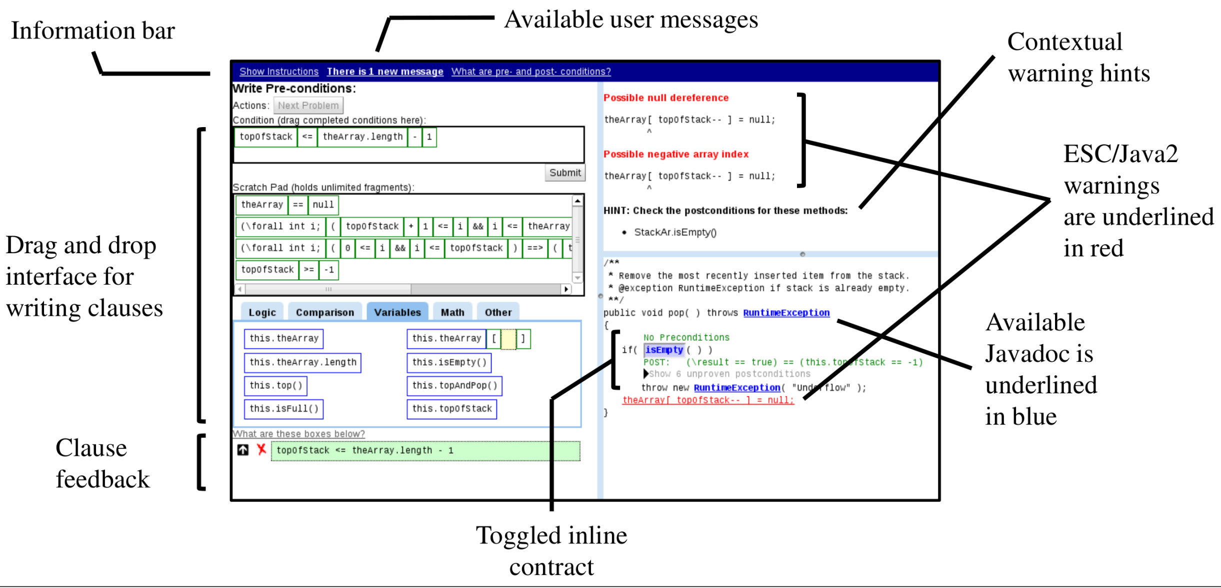
One example verification task is to prove that a program will never throw an unexpected null pointer exception, regardless of input. If a public method unconditionally dereferences a parameter, then the property is true only if the method is never called with `null` as an argument. The user can express this requirement by writing a method precondition that the parameter cannot be null. ESC/Java2 would then be able to prove that the method cannot throw an unexpected null pointer exception. However, the addition of the precondition introduces the requirement that the corresponding argument at each call site is non-`null`. In the end, ESC/-Java2 verifies that all the user-written contracts are mutually consistent, that the user-written contracts are consistent with the code, and that there are no null pointer exceptions. Note that, while this is partial verification, and not verification of full functional correctness, any requisite functional properties (e.g., data value properties) are verified *en passant*.

VeriWeb Use Case VeriWeb is targeted at the verification of domain-independent and expert-specified properties in existing client code. VeriWeb is not targeted at the discovery of new application-specific properties, verification of full functional correctness, the verification of library code, or the development of new programs.

VeriWeb focuses on client code rather than library code. We have observed that library code use is cross-cutting, whereas client code is more lightly coupled. Furthermore, the specifications of library code are complicated, in order to support general use — consider the C++ Standard Template Library. By contrast, the specifications for client code are relatively simpler, and the specifications required to show the absence of unexpected exceptions in client code are even simpler. Therefore, there is ample opportunity to reduce the cost of partial formal verification by focusing on client code.

Since VeriWeb operates by having users solve method subtasks, the tool is unlikely to uncover complex object invariants and global application properties. VeriWeb is better-suited to verifying language properties (e.g., that the program will never deference a null pointer) and localized application properties. This use case is aligned with the underlying tool ESC/Java2 [9], as well as similar techniques such as Microsoft's Code Contracts [8].

Evaluation To measure the time and monetary cost of verification, we performed a comparative study in which workers from Exhedra Solutions's vWorker [37] labor marketplace performed verification with ESC/Java2, either through its Eclipse interface or through VeriWeb. To understand the potential for using ad-hoc crowdsourced labor for verification, we recruited workers from Amazon's Mechanical Turk to use VeriWeb. To understand the overhead incurred when performing verification collaboratively with VeriWeb, we observed undergraduate students using the tool to collaboratively verify a small program.


Information bar
Available user messages
Contextual warning hints
ESC/Java2 warnings are underlined in red
Drag and drop interface for writing clauses
Available Javadoc is underlined in blue
Clause feedback
Toggled inline contract
Show Instructions
There is 1 new message
What are pre- and post- conditions?
Write Pre-conditions:
Actions:
Next Problem
Condition (drag completed conditions here):
Submit
Scratch Pad (holds unlimited fragments):
Logic
Comparison
Variables
Math
Other
What are these boxes below?

```
topOfStack <= theArray.length - 1

theArray == null
(\forall int i; ( topOfStack + 1 <= i && i <= theArray
(\forall int i; ( 0 <= i && i <= topOfStack ) ==> ( t
topOfStack >= -1

this.theArray              this.theArray [ ]
this.theArray.length       this.isEmpty()
this.top()                 this.topAndPop()
this.isFull()              this.topOfStack

topOfStack <= theArray.length - 1
```

```
Possible null dereference

theArray[ topOfStack-- ] = null;
       ^

Possible negative array index

theArray[ topOfStack-- ] = null;
       ^

HINT: Check the postconditions for these methods:

  • StackAr.isEmpty()
```

```
/**
 * Remove the most recently inserted item from the stack.
 * @exception RuntimeException if stack is already empty.
 **/
public void pop( ) throws RuntimeException
{
      No Preconditions
  if( isEmpty( ) )
      POST:   (\result == true) == (this.topOfStack == -1)
      Show 6 unproven postconditions
     throw new RuntimeException( "Underflow" );
  theArray[ topOfStack-- ] = null;
}
```

Figure 2. VeriWeb client interface showing a "write preconditions" subproblem. Top: information bar linking to instructions, FAQ, and messages about the problem (Section 2.2.4). Left: the drag and drop interface for writing conditions (Section 2.1.3). Right Top: ESC/Java2 warning locations are underlined in red in the source code view, and the warnings are shown at the top right. Lower right: source code view. Javadoc is available for code highlighted in blue. To view the documentation and warnings, the user simply hovers their mouse over the underlined code. The user has toggled the inline specifications (Section 2.2.3) for method isEmpty.

Contributions This paper makes 3 primary contributions:

1. We present novel verification interface features to address the challenges of decomposition and a distributed workforce. VeriWeb, a browser-based tool for program verification, incorporates these features.
2. We quantitatively characterize the time and monetary costs of verification.
3. We identify and describe challenges and threats to validity that are unique to studying verification in a global and collaborative setting.

The paper proceeds as follows. Section 2 describes VeriWeb's design principles and implementation. Section 3 poses three primary research questions. Section 4 quantifies the monetary cost of verification when contracting semi-skilled labor on an hourly basis. Section 5 explores the use of ad-hoc labor from Amazon's Mechanical Turk for performing verification. Section 6 characterizes the overhead incurred when performing collaborative verification with VeriWeb. Section 7 discusses the results. Finally, Section 8 presents related work, and Section 9 concludes.

2. VeriWeb

We created VeriWeb, a tool for verifying Java programs. Internally, VeriWeb uses the ESC/Java2 verification tool.[1] VeriWeb decomposes the task of verification into subproblems that users solve in a web interface (Figure 2). A live demo is available at the VeriWeb project page `http://www.cs.washington.edu/homes/tws/veriweb/`. We designed VeriWeb around two major principles: active guidance (Section 2.1) and explanations in context (Section 2.2).

2.1 Active Guidance

Active guidance — encouraging users to reason in a certain way, or restricting their set of actions — is aimed at aiding reasoning and preventing time-wasting mistakes. VeriWeb guides users *between* subproblems by choosing the next subproblem for users (Section 2.1.1) and *within* a subproblem by making suggestions (Section 2.1.2) and preventing syntax errors (Section 2.1.3).

2.1.1 Guided Decomposition

VeriWeb guides the user through the verification task by asking the user to solve 4 types of subproblems:

1. Select method preconditions from a list
2. Write method preconditions
3. Write method postconditions (that are true when the method exits normally)
4. Write method exceptional postconditions

[1] Other verification tools exist, such as Microsoft's Code Contracts [8]. We originally wanted to use Code Contracts for this research, but ESC/Java2 can statically reason about array properties that Code Contracts cannot. ESC/Java2 has its own disadvantages: it only fully supports the Java 1.4 specification (Java 5 was released in 2004), and it can be unsound.

VeriWeb offers the selection and writing of preconditions as separate problems not just because this focuses the user on the most relevant task at any moment, but also because they require different modes of reasoning. Selecting preconditions from a list of possibilities suggested by VeriWeb requires forward reasoning, to determine whether an error will occur. Writing preconditions is a more difficult task that requires backward reasoning from an error to determine the weakest precondition that will prevent it.

Active Subproblems A subproblem is *active* if it requires work from a user. There are three reasons that a subproblem may be active:

1. The subproblem is the cause of an ESC/Java2 warning. If there is a warning within a method body, the precondition set is considered to be the cause.
2. A user has identified the subproblem as being the cause of an error elsewhere. For example, an unverifiable postcondition might be caused by a too-weak precondition or by a too-weak postcondition on a callee method. Likewise for ESC/Java2 warnings occurring within the method body.
3. Every postcondition problem is initially marked as active. Users can often quickly write a few obvious postconditions (e.g., about the return value for a boolean method), and we found that doing so can substantially speed the verification process.

As long as any subproblems are active, the user is presented with an active subproblem to solve. When no subproblem is active, the program has been verified.

Subproblem dependencies VeriWeb computes dependencies among subproblems and stores these as a directed dependence graph that exhibits the same high-level structure as the call graph. Subproblem P_1 depends on subproblem P_2 if a change in the solution to P_2 may invalidate the solution to P_1. This property is independent of whether subproblem P_1 has actually been solved yet.

Whenever a user changes the solution to a subproblem, VeriWeb activates some or all of the subproblems that depend on it (Figure 3):

- If a precondition is strengthened, VeriWeb activates the precondition subproblems for the method's callers.
- If a precondition is weakened, VeriWeb activates the postcondition subproblems for the method.
- If a postcondition is weakened, VeriWeb naively activates all problems for the method's callers.

In each of these situations, VeriWeb re-checks each subproblem and only activates subproblems for which an error occurs. If no error occurs, then no additional work is required and VeriWeb does not require the user to re-visit the subproblem.

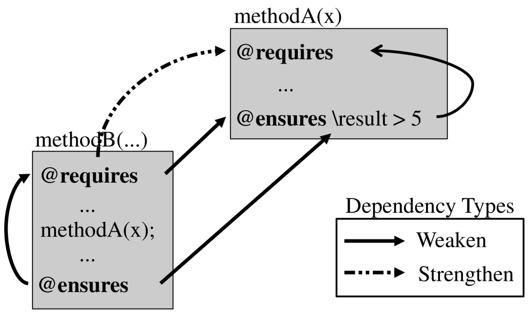


Figure 3. Intra- and intermethod depends-on graph for a method B that calls a method A. When a contract clause is weakened, subproblems with "weaken" edges leading to the contract are activated. For example, if the ensures clause for method A was weakened to `\result` > 4, both the precondition and postcondition problems for method B would be activated. When a contract clause is strengthened, subproblems with a "strengthen" edge leading to the contract are activated. For example, if a new precondition $x < 3$ is added to method A, the precondition subproblems for method B would be activated to ensure the call to method A uses a valid x value.

Note that these basic rules capture more complex relationships between properties via composition. For example, according to the second activation rule, if a method's preconditions are weakened, VeriWeb will activate the postcondition problem for that method. If the change causes the postcondition set to be weakened, then VeriWeb will activate all the problems for the method's callers according to the third activation rule.

The leaves of the active subproblem graph — those nodes that are active and have no active children — are the set of subproblems that can be productively assigned to workers. In fact, VeriWeb can assign these subproblems to be solved in parallel by different workers. When assigning a subproblem, VeriWeb gives preference to users who have already worked on a subproblem, and to users who have marked that a specification was incorrect.

Collaborative Use To support multiple users working simultaneously, VeriWeb currently just naively performs synchronization on an entire project. Each user keeps a local view of the master project specification; the local view is updated whenever the user requests a new subproblem.

When a user submits a solution to a subproblem P_1, VeriWeb checks whether the solution invalidates the assumptions of any other subproblem P_2 currently being worked on. VeriWeb lets the other worker continue working on P_2. However, when the worker submits P_2, VeriWeb records the clauses in the solution for future use, but does not modify the master specification.

Recursion The current implementation of VeriWeb only supports single-method recursion. It is straightforward to extend VeriWeb to mutually recursive methods: make a tree of the (possibly degenerate) strongly connected components (SCCs), and arbitrarily choose one method from each SCC that is a leaf. Once the user solves this subproblem, the SCC is either broken or is smaller.

Object Invariants Object invariants require non-modular reasoning about all public methods in a class, which is at odds with VeriWeb's decomposition into subproblems. VeriWeb does not display object invariants nor give the user the opportunity to write object invariants directly.

VeriWeb does, however, compute and utilize object invariants by lifting conditions that appear as preconditions and postconditions for all the public methods for the type. The tool expedites object invariant discovery in the following ways:

- Any precondition that refers to object state is automatically checked as a (potential) postcondition for the method. (Section 2.1.2 discusses how VeriWeb suggests and checks potential contracts.)
- Any method invariant — a clause in both a method's pre- and postconditions — is automatically suggested in precondition selection problems for other public methods in the class, and is automatically checked as a (potential) postcondition for other public methods.
- When a clause has been established as a method invariant for at least half of the non-pure public methods in the class (those methods annotated as not mutating state), the interface prompts the user to indicate whether or not the clause is an object invariant; if the user agrees, the condition is added as a precondition to the other methods and subproblems are activated as previously described.

This scheme for handling object invariants is far from perfect; non-trivial object invariants should by written directly by the feature developer.

2.1.2 Contract Suggestions

Writing contracts from scratch is difficult, especially for users unfamiliar with the specification language. VeriWeb generates, and presents to users, a suggested set of possible clauses. Users can often complete their task just by selecting clauses; in other cases, users can select some clauses and write others.

Contract suggestions convey two kinds of benefits. First, it is much faster, and requires less creativity, to select clauses rather than writing them from scratch. Second, the inferred clauses serve as a model when writing new contracts: they can both illustrate the syntax in which clauses are written and, even when slightly incorrect, can inspire users to write similar clauses. We have observed all of these benefits.

VeriWeb makes suggestions for both pre- and postconditions; however, the list of subproblem types (Section 2.1.1) includes "select preconditions" but not "select postconditions" because VeriWeb automatically performs all postcondition "selection." When postcondition problems are presented to the user, VeriWeb queries ESC/Java2 regarding each suggested postcondition, displays them all, and indicates which ones are verifiable and which ones are not verifiable; the user may hide the unverifiable ones.

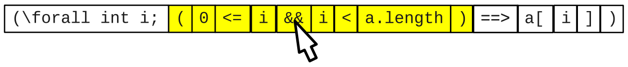

```
(\forall int i; ( 0 <= i && i < a.length ) ==> a[ i ] )
```

Figure 4. Subfragment highlighting, shown here in the drag and drop interface, helps users read and understand clauses.

There are multiple ways to generate suggested clauses. The current VeriWeb implementation uses Daikon [16, 17] to dynamically infer likely clauses from execution traces. Because the suggested clauses generalize observed executions, they may not be true; even if true, they may be beyond ESC/Java2's ability to verify them. However, Nimmer and Ernst [28] found that (1) users annotating programs are not encumbered by false inferred clauses, and (2) for two of their three subject programs, including inferred clauses had a statistically significant positive correlation with successful verification.

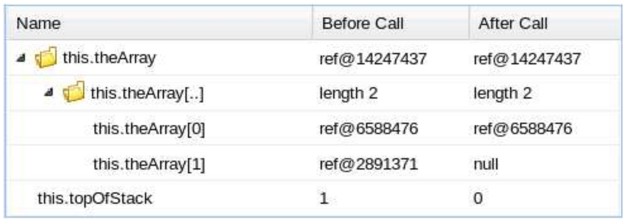
2.1.3 Drag and Drop Contracts

We observed that many users wasted time attempting to write invalid clauses — clauses that were either syntactically or semantically incorrect. For example, some users attempted to write facts about local variables in a method's preconditions. To guide users in writing expressions, we developed a drag and drop interface to allow users to construct clauses from a pool of starter "fragments." Invalid fragments, such as the `\result` and `\old` fragments when writing preconditions, are not available. Fragments have holes where other fragments can be inserted or removed. Contracts displayed in documentation can be added to the drag and drop interface by clicking on a button next to the contract.

Large contracts and fragments, especially those involving complex constructs such as universal quantification, are difficult for humans to read. VeriWeb's subfragment highlighting (shown in Figure 4) enables users to better understand the subexpression groups. Additionally, the highlighting indicates the subfragment that will be removed when the user clicks and drags. Subexpression highlighting is also enabled for contracts displayed in other parts of the interface.

2.2 Explanations in Context

Program verification tools tend to be inscrutable. After a verification failure, it can be difficult for users to understand the tool's internal state or reasoning steps, and to know what changes would permit a specification to be verified. Users are often frustrated as they try to form and maintain their own mental model of what the tool knows and/or can prove. VeriWeb offers clues to make this work easier or to eliminate it entirely. VeriWeb explains the relationship of elements *within* a subproblem with concrete counterexamples (Section 2.2.1) and tool tips (Section 2.2.2), and *between*



Name	Before Call	After Call
this.theArray	ref@14247437	ref@14247437
this.theArray[..]	length 2	length 2
this.theArray[0]	ref@6588476	ref@6588476
this.theArray[1]	ref@2891371	null
this.topOfStack	1	0

Figure 5. VeriWeb clauses are "executed" over a dynamic trace. If the clause is falsified by the trace, the parameter and field values before and after the call are displayed in an expandable tree grid.

subproblems with contract inlining (Section 2.2.3) and intermethod dependency information (Section 2.2.4).

2.2.1 Concrete Counterexamples

A typical program verification tool indicates that a given clause is either provable or unprovable. Given an unprovable clause, a user does not know whether the contract is false, the clause is true but beyond the reasoning abilities of the verification tool, or the clause would be provable if other parts of the specification were improved. We observed users spending significant amounts of time fruitlessly attempting to prove false clauses. To eliminate this wasted effort, VeriWeb presents the user with concrete counterexamples for clauses that are demonstrably false with respect to a set of real executions (e.g., from running the test-suite). VeriWeb does not currently generate concrete executions or tests; these are provided by the feature developer.

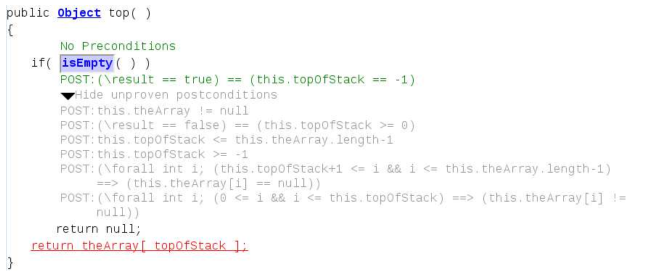
VeriWeb displays the counterexample information in two ways. First, when the user hovers the mouse pointer over a subexpression of the clause, a tooltip shows the value of that subexpression. Second, the user can explore all the parameter and return values via an expandable tree grid (see Figure 5). Expanding the grid shows the fields of each object.

Our run-time checking currently has two limitations: First, no testing is done of conditions in exceptional postconditions (because our trace generator does not handle exceptional exits). Second, expressions involving universal quantification can be tested only if the quantified-over variable has explicit lower and upper bounds.

2.2.2 Task-specific Tooltips

Contextual help messages are weaved into the interface. For example, in the drag and drop interface, hovering the mouse over a hole in a `\forall` expression shows what type of expression should be placed there (e.g., a predicate to bound the quantified variable).

Other examples of contextual help messages include tips shown between problems (while the next problem is loading), FAQ links displayed above and below interface elements, and additional tooltips.



```
public Object top( )
{
        No Preconditions
    if( isEmpty( ) )
        POST:(\result == true) == (this.topOfStack == -1)
        ▼Hide unproven postconditions
        POST:this.theArray != null
        POST:(\result == false) == (this.topOfStack >= 0)
        POST:this.topOfStack <= this.theArray.length-1
        POST:this.topOfStack >= -1
        POST:(\forall int i; (this.topOfStack+1 <= i && i <= this.theArray.length-1)
             ==> (this.theArray[i] == null))
        POST:(\forall int i; (0 <= i && i <= this.topOfStack) ==> (this.theArray[i] !=
             null))
       return null;
    return theArray[ topOfStack ];
}
```

Figure 6. Contract inlining: contracts for a method are aligned with the method in the source code. Users can toggle the display of unproven postconditions. Inlining the specs for multiple methods can help to identify "information gaps."

2.2.3 Contract Inlining

When verifying a program, information about the absence of knowledge (e.g., "Why doesn't the tool have enough information to prove this postcondition?") can be as important as information about what the tool does know. To help users locate "information gaps" between method calls, VeriWeb offers contract inlining in the source view. Contract inlining displays a method's contracts around a method call in the source code: the preconditions are above, the postconditions are below, and everything is horizontally aligned with the method call. An example is shown in Figure 6. The inlined contracts currently provide two pieces of additional information: (1) the set of preconditions that are not met at the call site and (2) a user-toggleable list of unproven (potential) postconditions for the callee method.

Initially, no contracts are inlined; the user can display as many or as few as desired. Contract inlining even works for source lines with multiple method calls: the inlined contracts are displayed from outside to inside in the order that the calls appear on the line (i.e., horizontally aligned with the corresponding call). Contracts can be verbose and might clutter the display, but we hypothesize that users only need to inline contracts for 2–3 methods at a time (to visualize the information gaps). Therefore, our approach should scale even to methods that make many method calls.

2.2.4 Intermethod Dependency Information

Postcondition Dependencies We observed that it is difficult for new users to form and maintain a mental model about what the verifier knows after a method call. One ramification of this is that the users do not realize that warnings in a method may be caused by deficient postconditions for the method's callees. Contract inlining (Section 2.2.3) addresses this problem. To further address it, when VeriWeb displays verifier warnings, it also lists the methods that are called before the warning and indicates that the user may need to refine the postconditions of those methods. While data flow analyses could be utilized to make this information more precise, pilot tests showed that users still found these messages helpful.

Information Transfer Because methods depend on one another, some information must transfer between subproblems. In VeriWeb, when a user assigns blame (e.g., marking that a callee's postconditions are too weak), the user is asked to explain the problem either in English or pseudo-code.

Relevant messages from other subproblems are displayed with the current subproblem. The user completing the newly created task can mark whether or not the comment was helpful. If the user indicates that the comment was not helpful, the user is prompted to explain why they did not find the comment helpful. In the future, we plan to use this feature to automate the handling of payments when deploying VeriWeb on an ad-hoc marketplace such as Mechanical Turk [1].

3. Research Questions

To validate the VeriWeb tool, as well as the general potential for crowdsourced program verification, we posed three research questions:

RQ 1. *What is the cost (time and money) of program verification?*

To answer this question, we performed a comparative study of VeriWeb and the ESC/Java2 plugin for Eclipse (Section 4). The subjects were 14 programmers recruited from Exhedra's vWorker labor marketplace. We found that the VeriWeb workers took both less time and money on average than the Eclipse workers for the subject program. Additionally, we found that worker progress with VeriWeb was more consistent than with Eclipse, which was characterized by work toward incorrect solutions and oscillations around local optima.

RQ 2. *Can ad-hoc labor be used to crowdsource program verification?*

To explore this question, we recruited workers from Amazon's Mechanical Turk at varying pay levels to complete VeriWeb subproblems (Section 5). We found the labor pool to be very shallow; additionally, the expected pay was not competitive with that of hourly contracted workers (c.f. Section 4). The preliminary results suggest that current ad-hoc labor marketplaces are not well-suited for verification.

RQ 3. *How does decomposition and communication overhead affect the performance of collaborative verification?*

To characterize the overhead, we observed two pairs of computer science undergraduates each using VeriWeb to collaboratively verify a small program (Section 6). We found that, for each pair, VeriWeb generally isolated one participant from the other — for one pair, neither participant observed any communication from the other. Additionally, the observations highlighted that while collaboration can speed verification by enabling users to address a root cause in parallel, a single poor-performing user can significantly derail verification.

Pilot Studies During the development of VeriWeb, we performed pilot studies similar to the first study above. These studies were performed with both hourly contracted workers and more than 40 computer science undergraduate students.

4. The Cost of Program Verification

Creating a cost- and time- effective tool requires solving an optimization problem [31]. Though the problem involves many complicated factors, an approximation consists of only two complementary questions:

RQ 1.1. *Given a fixed amount of money, how "much" verification can you buy?*

RQ 1.2. *Given a fixed amount of time, how "much" verification can you buy?*

In this section, we begin to quantitatively answer both of these questions by contracting workers on Exhedra Solutions's vWorker [37] marketplace to verify a small project `StackAr`, which we consider to be of moderate difficulty based on prior work [28].

vWorker Labor Marketplace The vWorker [37] marketplace boasts a global workforce of over 320,000 registered workers, over 150,000 registered employers, and 1,500 – 2,500 projects in open bidding at any given time. Employers typically post small to medium business projects ($50 – $1000), including design and programming jobs, for workers to bid on in an open auction (invite-only auctions are also possible). The employer then selects a worker(s) to work on the project. For its role as a matchmaker and arbiter, vWorker takes a 7.5% – 15% cut from worker pay. Several sites offer services similar to vWorker. We chose vWorker due to the first author's positive experience with the service in the past.

4.1 Experimental Design

We posted a project with the headline "Write Java program specifications" on vWorker. The project posting had users place hourly bids for up to 6 hours of work and stated that we would accept multiple bids. We placed no restrictions on the workers. (vWorker lets you accept bids only from workers in developed countries, or mandate that the worker use a webcam so that you can monitor their work.)

Subject Program We used the `StackAr` program [39], an implementation of an array-based stack, from a previous study of program verification [28]. The program consists of a data type (library class) paired with a test program that throws run-time exceptions if certain correctness properties do not hold. For example, the client checks that a call to `isEmpty()` returns `true` for a new stack.

The `StackAr` class has 8 methods, consisting of 49 NCNB lines of code. The client class consists of 79 NCNB lines of code. Using ESC/Java2, 23 JML annotations are necessary to show that the data type and client suffer no unexpected run-time exceptions.

We added clauses and annotations asserting the types of the objects and arrays so that we did not have to teach the participants the type syntax of JML (these clauses were trivial, and were automatically inferred by Daikon).

The Daikon-inferred clauses, which were offered to both VeriWeb and Eclipse users (see below), came from Nimmer's client code, which used the library in realistic ways [28]. By contrast, VeriWeb's counterexamples came from the included ADT clients, which are (rather impoverished) example uses. As a result, the VeriWeb counterexamples did not add any additional information beyond what the Eclipse users could learn by inspecting the included ADT clients.

Treatments We used the ESC/Java2 plugin for Eclipse 3.5[2] as the "standard" user interface. VeriWeb suggests possible clauses inferred by Daikon, so to level the playing field, we inserted those same the inferred specifications in the subject programs used by Eclipse. Both interfaces were hosted on a group of Rackspace Cloud server instances (Eclipse was served via Windows terminal services). Progress in Eclipse was logged by recording each text edit, and recording when the user invoked ESC/Java2 to check the project.

Participants We received 26 official bids ranging from \$6/hour to \$110/hour (Mean: \$21.8/hour; Median: \$14.5). For the experiment, we accepted the 18 bids in the \$6/hour – \$22/hour range and split the participants into treatment groups to produce roughly equivalent rate distributions. Complete bid data, including worker country, can be found on the project website.[3]

Worker Skill We expected that assigning the groups based on requested pay would result in groups with similar programming skill distributions. We asked each worker both their programming and Java experience.[4] Figure 7 shows that workers' bids do not correlate with experience, placing into doubt our initial assumption that requested pay is a viable proxy for skill.

Instructions Each worker was given a web link to a description of JML [6, 22] contract syntax and instructions for how to run their respective tool. Workers were instructed to spend approximately 1 hour on a warm-up tutorial. Upon completing the tutorial, they were given a quiz about JML specifications and their tool to ensure comprehension. For each worker's initial quiz attempt, we provided a list of questions answered incorrectly, references to the sections containing the answers to the questions, and had the workers correct their answers before continuing with the main task. Aside from one worker, all of the workers answered at least one question incorrectly on their initial quiz attempt. 9 out of the 14 workers finishing the project reported spending longer than an hour on the warmup. A deadline of one week was given for the project; though this deadline was not enforced, it may have contributed to the 4 workers who did not complete the project.

[2] `http://secure.ucd.ie/products/opensource/ESCJava2/`

[3] `http://www.cs.washington.edu/homes/tws/veriweb/`

[4] We asked for this information after the experiment. Asking before might have given workers a motivation to lie, in order to obtain a higher wage.

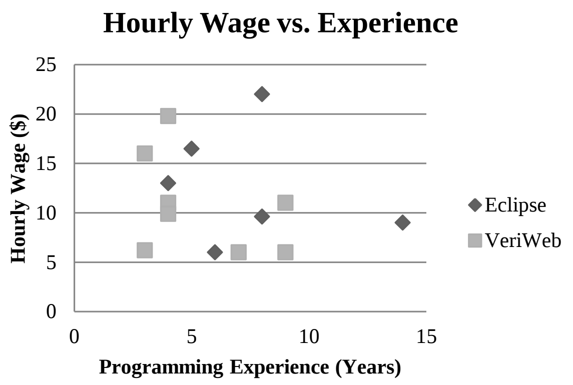


Figure 7. Requested hourly pay (bid) versus programming experience reported upon completion of the project. Workers' bids were not an accurate proxy for programming experience.

Edit Distance Performance Metric The number of warnings reported by ESC/Java2 is not an accurate measure of progress, as one incorrect annotation can mask other warnings. Therefore, we instead use *edit distance*, the minimal number of additions and removals of top-level JML annotations that yields a known verifiable solution. This is a lower bound on the amount of work required to complete the specification task. (Actually, we use a modified version of the metric that accounts for the differences between the tools. See the Appendix.)

A verification problem has many possible (legal) solutions. The target solution set included solutions from [28], our pilot studies, and the solutions discovered during the study themselves. Whenever a solution included an object invariant, we added another solution in which the object invariant was explicitly listed as method conditions. The edit distance is the distance from a worker's current version of the program, to the nearest of any of these legal solutions.

To efficiently calculate distance to verifiable specifications, we wrote a tool to normalize the specifications and perform a textual comparison. The normalization rewrites constant subexpressions and inequalities, uses De Morgan's law to split conjunctions (e.g., `P ==> Q && R` is split into `P ==> Q` and `P ==> R`), etc.

4.2 Results

8 VeriWeb workers and 6 Eclipse workers completed the project; the other workers (3 Eclipse workers, and 1 VeriWeb worker) would not comment on why they dropped out of the project. Figure 8 shows the distance to the nearest verifiable

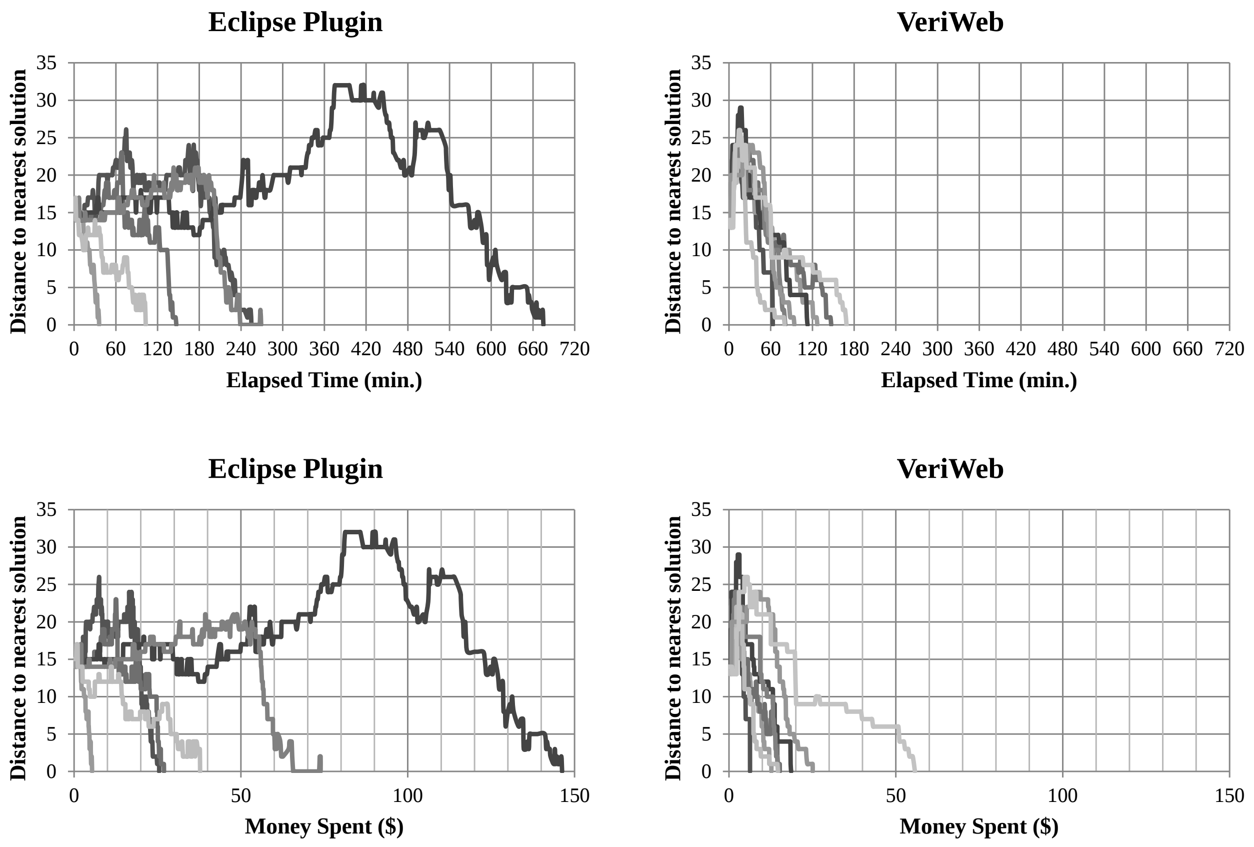


Figure 8. Top: progress as a function of time. Bottom: progress as a function of money spent. Data series (line) coloring is consistent between the graphs, e.g., the darkest lines in both Eclipse graphs both correspond to the same worker.

solution for the workers as both a function of time and money spent. The graphs illustrate three salient features.

First, on average, the VeriWeb workers completed the project faster and for less money than the workers using Eclipse. However, the relatively small number of workers in the study prevented the rejection of both the null-hypothesis that the completion time distributions are the same and the null-hypothesis that the money spent distributions are the same (the one-tailed Mann-Whitney U-test p-values are 0.0985 and 0.0694, respectively). If the Eclipse worker with 14 years of programming experience — an American who only charged $9/hr — is excluded from the sample, the differences are both significant (one-tailed p-values of 0.0202 and 0.0116, respectively). The variance in completion time is smaller for VeriWeb. This is promising because one main goal of VeriWeb is to commoditize verification, and low variance (predictability) and commoditization go hand in hand. However, the null-hypothesis that the variances are equal cannot be rejected when using the non-parametric Brown-Forsythe Levene-type test (p-value is 0.0725).

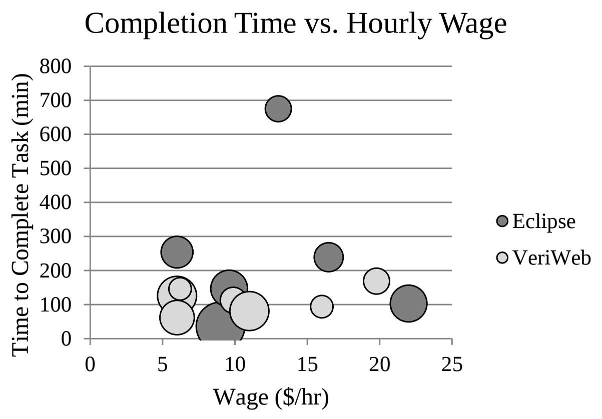
Second, the VeriWeb workflow is characterized by continual progress with intermittent back-tracking. By contrast, the Eclipse workflow is characterized both by work toward incorrect solutions and by oscillations around local minima. As with the reduced variation in completion time, this is encouraging with respect to commoditization.

Third, while the total cost of verification is linearly correlated with total time for Eclipse ($r = 0.946$), the correlation is lower for VeriWeb ($r = 0.694$) indicating that different workers should be chosen whether you are optimizing for time or money.

4.2.1 Effects of Skill

Figure 9 shows the time taken to complete the task given the workers' hourly rates and reported programming experience. As previously noted, the rate is likely not a good proxy for skill, as it was not correlated with worker experience. Multiple factors may confound the relationship between bids and skill.

- Variations in the cost of living and exchange rates between countries, and within countries, may cause the assumption to be violated if the geography of the workforce is diverse. We adjusted the bids based on an OECD report on purchasing power parity [30], but this still did not account for the differences (due to *intra*national bid variation).



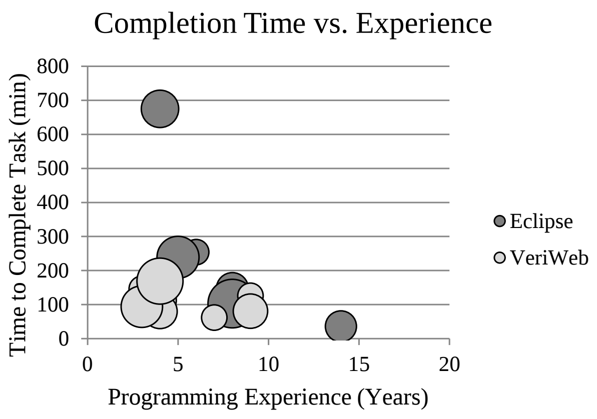


Figure 9. Left: time to complete the task as a function of hourly wage. Bubble size indicates the worker's relative programming experience. Right: time to complete the task as a function of the worker's self-reported programming experience. Bubble size indicates the worker's relative hourly wage. There is no relationship between wage and productivity for either tool — this is likely explained by the absence of a relationship between requested hourly wage and programmer experience (see Figure 7).

- vWorker employs a worker rating system. Workers with higher ratings may demand a premium; skilled workers with a short work history may proffer a low bid in order to establish a good rating.

4.2.2 Feature Usage

The 14 successful workers took a short survey about their tool's specific features.

Suggested Conditions Contrary to what we found in pilot studies, the VeriWeb users only found the suggested conditions moderately helpful (mean of 3 5/8 on a 5-point Likert item), due the to inclusion of incorrect or irrelevant conditions. One user also felt that some of the conditions took too much effort to understand. The Eclipse workers found the suggested conditions more helpful (mean of 4 1/3 on a 5-point Likert item).

Drag and Drop Interface Three of the users preferred the drag and drop interface, four users preferred the text interface, and one user used both depending on the complexity of the condition (preferring it for complex conditions). As expected, the drag and drop interface was preferred for not having to worry about incorrect syntax; text entry was preferred for speed.

Contract Inlining All of the VeriWeb users reported using the toggle-able contract inlining to solve the task (however two users' responses indicate that they misunderstood the feature to which the question was referring). Users reported using the feature to (1) remember conditions for methods that had already been visited, (2) identify missing postconditions, (3) view multiple method specifications simultaneously, and (4) understand unexpected exceptions.

Counterexamples VeriWeb generated concrete counterexamples for 4 of the 24 inferred clauses provided to the workers. While VeriWeb also prevented users from writing additional clauses that violated the trace, Eclipse did not — the successful Eclipse workers introduced an additional 26, 13, 9, 9, 3, and 1 falsifiable clauses when working on `StackAr`. The distribution of these clauses' lifetimes were highly skewed with a mean of 34 minutes, and a median of just 2 minutes. Unsurprisingly, the Eclipse user who introduced 26 falsifiable clauses took the longest to complete the task. For 2 of the users, the number of falsifiable preconditions met or exceeded the number of falsifiable postconditions introduced. Falsifiable preconditions are especially harmful because users "propagate" the false preconditions to the method's callers.

VeriWeb provided counterexamples to each of the workers, however only four of the eight workers reported being aided by counterexamples. One user reported using the feedback to identify an off-by-one error. Another user reported using the feedback to identify a missing condition for a callee.

4.3 Discussion

Feature evaluation Directly measuring the effect of each VeriWeb feature on performance would have required an intractably large number of subjects. The survey responses and quantitative results indicate that each of the features contributed to the usability of VeriWeb. Feature usage metrics suggested that certain features, especially concrete counterexamples, are good first candidates for individual evaluation.

Threats to Validity Due to the small study size and single subject program, the performance characteristics observed may not generalize. In particular, the following caveats apply:

- VeriWeb's speed is bounded by ESC/Java2's speed, and ESC/Java2 does not scale.
- The subject program is array-based, whereas much Java code is collection-based. We opted to not use a subject program with collections because the ESC/Java2 specifications for collections are fragile (often resulting in unsoundness) and significantly slow the checker.

More general threats to validity are discussed in Section 7.3.

5. Verification with Ad-hoc Labor

In this section, we explore the (lack of) ad-hoc verification labor supply on Amazon's Mechanical Turk marketplace [1] for performing VeriWeb subproblems.

5.1 Mechanical Turk Labor Marketplace

Amazon's Mechanical Turk [1] is an online marketplace where employers can post human intelligence tasks (HITs) to be solved by a global ad-hoc workforce. HITs are typically small tasks that are easy for a human to perform but difficult to automate. Common tasks include image labeling, preference surveys, and audio transcription.

5.2 Experimental Design

For our experiment, we created a batch of 50 HITs with the title "Answer questions about Java methods" and description "Describe what must be true when a method is called and after it runs." Each HIT required the worker to complete at least three VeriWeb subproblems (guaranteeing that each worker solved at least one precondition problem); workers could complete additional subproblems within the HIT for additional pay.

Subject Program The subject program was the array-based stack, `StackAr`, previously described in Section 4. To provide a consistent VeriWeb experience for each worker, each worker worked on their own copy of the project. Workers did not work collaboratively with other users. The subproblems presented to a worker followed the methodology described in Section 2.1.1.

Variable Pay Mechanical Turk permits requesters to embed external websites within a HIT, informing the website whether the HIT is in preview mode, or has been accepted by the worker. We used this information to allow the workers to try using VeriWeb before accepting the HIT.

Typical HITs pay a fixed rate, however, the service offers the ability to pay a bonus for good work. Combined with the preview feature, we use this functionality to randomly pay a different rate to each worker (between \$0.15 and \$0.35 a subproblem). We set the static pay rate of the HIT to \$0.00 and appended the phrase "BONUS PER QUESTION" to the HIT title and description. The preview screen for the HIT informed each worker the amount that they would be paid for solving each subproblem. Browser sessions were used to ensure that each worker only saw a single price, even when previewing the HIT multiple times. The VeriWeb interface was modified to display the amount of money the worker had earned so far in the top information bar.

Learning Curve VeriWeb's steep learning curve relative to other tasks on Mechanical Turk is a major obstacle. We opted to not provide a formal tutorial, instead relying on VeriWeb's contextual help (see Section 2.2.2); additionally, the first subproblem was chosen to be trivially easy — a "select requires" problem consisting of a single choice sufficient to eliminate all warnings — to encourage acceptance.

5.3 Results and Discussion

Ideally, the results of the experiment would enable the creation of a labor supply curve: the number of subproblems solved vs. pay (per subproblem). However, worker response was unenthusiastic — fewer than 10 workers accepted the HIT over the course of 3 days. Workers required at least \$0.25 per subproblem to complete any subproblems. No Mechanical Turk worker completed more than 5 subproblems, indicating that the rate would have to be further increased to account for problem difficulty. We elected not to perform the study again with higher rates, as the effective hourly rate would not have been competitive with the hourly rate for workers contracted via vWorker (see Section 4).

Threats to Validity In addition to a possible lack of generalizability due to the use of the single subject program `StackAr`, the following three factors likely affected the observed result:

- The absence of a fixed price for the HIT reduces views of the HIT by workers who search for HITs by payment range.
- The HIT is unique and unusual. Workers are unlikely to "invest" in learning to perform a HIT if they perceive the opportunity to perform similar tasks in the future is low. Conversely, the novelty of the HIT may attract some workers that would not normally work on the HIT.
- Labor supply may vary throughout the week. However, allowing a HIT to run for a whole week would be suboptimal — as the HIT ages, its position in the "latest HIT" list lowers, decreasing the chance that it will be seen by a worker.

Despite these threats to validity, it is clear that VeriWeb (or the presentation of VeriWeb) is insufficient to address the ease-of-use and profitability demands of the Amazon Mechanical Turk labor pool.

6. Overhead of Collaborative Verification

Since VeriWeb decomposes the task of writing a verifiable specification into subtasks, multiple workers can work on the task simultaneously. For programs with limited coupling (modulo library code), workers can largely work in parallel. To characterize the overhead incurred when workers work on

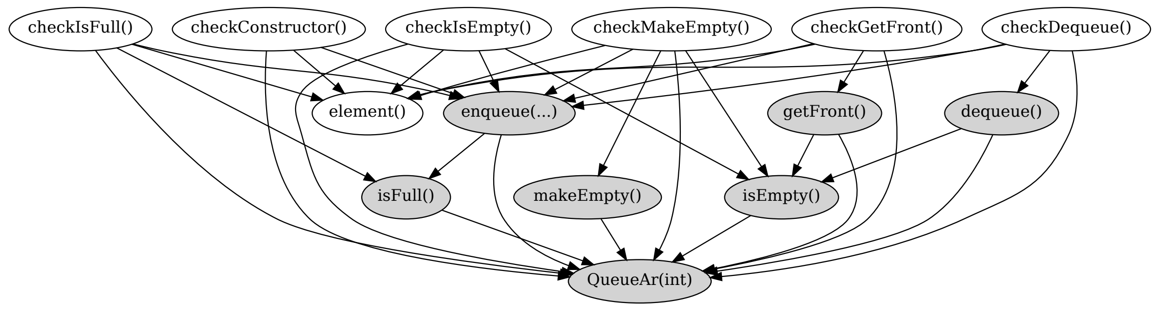


Figure 10. Method dependency graph for the subject program in the collaborative study (Section 6). The queue data type classes are shown in gray, the client classes are shown in white. The program has many methods for which the subproblems can be performed in parallel. For example, once the QueueAr(int) subproblems are solved, workers can simultaneously work on the subproblems for isFull(), makeEmpty(), and isEmpty().

interdependent subtasks, we observed five computer science undergraduate students using VeriWeb to individually and collaboratively verify a queue data type and client class.

6.1 Experimental Design

Participants The study participants were computer science undergraduate students at the University of Washington with 3–5 years programming experience. Use of undergraduates was intended to emulate the use of semi-skilled workers.

Treatments We observed three treatments. For each treatment, the participants first (individually) performed a tutorial, and then verified the subject program.

- Treatment 1: a single user performed the tutorial, took and received feedback on a comprehension quiz, and then verified the subject program.
- Treatment 2: two users each individually performed the tutorial, and then collaboratively verified the subject program without the assistance of dynamic counterexamples.
- Treatment 3: two users each individually performed the tutorial, individually took and received feedback on a comprehension quiz, and then collaboratively verified the subject program.

Treatment 2 was performed before the other treatments. The lack of dynamic counterexamples for Treatment 2 was due to a tool setup error. The quiz from the study in Section 4 was added in light of the Treatment 2 results (see Section 6.2) — we wanted to separate the effects of learning and tool comprehension from the effects of collaboration.

Subject Program We used the `QueueAr` program [39], an implementation of an array-based queue, which was also used in a previous study of program verification [28]. Like `StackAr` in the comparative study (Section 4), the program consists of a data type (library class) paired with a test program that throws a run-time exception if certain correctness properties do not hold. Figure 10 shows the dependency graph for the subject program; we selected this subject program because it exhibits a small amount of parallelism suitable for the number of workers in the study.

6.2 Results

Completion Time The individual participant (Treatment 1) with 2.5 years of programming experience completed the task in 52 minutes. For reference, Nimmer and Ernst had previously reported that none of their 41 participants could complete the task within a 60 minute period (participants were stopped after an hour) [28].

In both Treatment 2 and Treatment 3, the participants worked in parallel on subproblems containing warnings caused by a single root cause — a missing exceptional postcondition for the `enqueue` method. For both treatments, these subproblems were the most difficult for the participants to solve.

The pair in Treatment 2 completed the task in 54 minutes (wall clock time). 20 of the 54 minutes was spent assessing the missing exceptional postcondition: one participant spent 20 minutes studying the `checkConstructor` method; the other spent 10 minutes on the `checkIsEmpty` method. As both participants had stalled, we asked the participants to explain their reasoning out loud. Based on this information, we reminded the participants how to mark a callee's exceptional postconditions as insufficient. Both participants were then able to continue.

The pair in Treatment 3 had not completed the task after 55 minutes (wall clock time), but could not continue as a user input caused the tool to crash. One participant correctly added an exceptional postcondition to `enqueue`, but this was insufficient as the `isFull` method was missing a postcondition about its return value. The other participant then began needlessly strengthening method preconditions in an attempt to avoid the exception, ultimately triggering a combination that caused the tool to crash.

Messages During the course of the task, neither participant in Treatment 2 worked on a subproblem that had associated messages from the other participant. In Treatment 3, one of the participants worked on a problem with a message from the other participant that consisted of two syntactically correct JML clause suggestions. Since the tool does not attempt to parse messages, the participant saw the message as just an extra step to adding the missing conditions.

The lack of direct communication was a result of VeriWeb's rules for assigning subproblems, which prefers workers with knowledge of the subproblem (see Section 2.1.1). Additionally, for Treatment 2, the participants did not introduce incorrect specifications that required multiple levels of backtracking.

6.3 Discussion

The results highlight a major benefit of collaborative verification with VeriWeb: workers can simultaneously work on the same underlying issue. If one worker gets stuck, another worker might be able to identify and solve the issue by approaching it from a different perspective. The Treatment 3 results also expose a major drawback of collaborative verification with VeriWeb: a single worker can derail the verification process by solving subproblems incorrectly.

For the subject program, VeriWeb's problem preference system performed well for presenting a consistent workflow to the individual users. Therefore, future research on the efficacy of the message system must still be performed, as the opportunity for interaction increases as the number of workers and size of the call graph increases.

Experimental Design Having the participants work side by side conveyed the benefit of a single observer being able to observe the workers simultaneously. Unfortunately, the setup precludes having the participants narrate their thought process. Additional studies should have multiple observers observing the participants in isolation to enable narration.

Threats to Validity In addition to the threats described in Section 4.3, the following threats to validity exist:

- The results may not scale to a larger number of workers working simultaneously due to increased interaction.
- All of the participants were native English speakers, and the documentation and variable names were in English. Workers with limited ability to read / write a common language are likely to incur greater overhead when communicating with other workers.
- The participants had all completed the same core computer science curriculum. In practice, workers may not have shared computer science training, or formal training at all.

Overall, we believe the benefit of being able to simultaneously observe the participants in person outweighed the threats to validity caused by our participant selection.

7. Discussion

7.1 Crowdsourced Verification in Practice

Our primary study (Section 4) explored the hourly pay model for verification in a global marketplace. Such a model lies in the middle of the spectrum between ad-hoc labor marketplaces like Mechanical Turk (Section 5) where work engagements are fleeting, and contract labor that is typical of more traditional approaches such as outsourcing. It remains to be seen under what conditions the different positions on the spectrum are optimal for program verification and other software engineering tasks.

In any case, verification and specification experience is scarce. Therefore, for the time being, employers that opt to use crowdsourcing must decide whether to pay a premium for workers with formal methods experience or to "train" new users to use the technology, as we did in the vWorker study. The most cost-effective approach is most likely to meet the workers half-way: building tool support to decrease the skill demands of the task, and, at the same time, offering targeted training to set the workers up for success.

7.2 Validity of Specifications

VeriWeb generates a specification that is sufficient to prove the lack of runtime exceptions; the "correct" or intended specification is unknowable without input and validation from the feature designers. However, there are three ways that VeriWeb can be led to the intended specification:

1. Informal documentation (Javadoc)
2. Checks in the implementation (e.g., checking of preconditions and throwing an IllegalArgumentException)
3. The feature designer can write JML specifications for high-level properties

Approaches #1 and #2 can be used by VeriWeb with no extra effort from the feature designer. Approaches #2 and #3 induce verification criteria that lead VeriWeb to find the "correct" specification, while Approach #1 depends on the workers taking cues from the documentation.

7.3 Threats to Experimental Validity

In addition to the experiment-specific threats enumerated in Sections 4, 5, and 6, the following threats to experimental validity apply.

Learning Effect Workers may perform better because they acquire more experience (a "learning effect"). For the studies in Section 4 and Section 6, we employed a mandatory warmup and quiz to mitigate the learning effect.

Program Correctness All of our subject programs are "correct" in the sense that ESC/Java2 can verify lack of run-time exceptions without modifying the source code. The performance and mindset of workers likely depends on (1) whether the program is correct, and (2) whether they believe that the program is correct. We did not tell the vWorker and Mechanical Turk workers that the programs were correct or incorrect. The vWorker workers might have inferred this from the task description. It is possible that (accurate) belief that a program is correct might qualitatively change the way that the workers would work.

Future research should investigate workflows for verification when software modifications are necessary, such as for fixing bugs or improving design. An intuitive workflow for dealing with software bugs would be to escalate areas that cannot be verified to software developers (possibly escalating to more-skilled program verifiers first).

Program Complexity Real programs often contain poor design: complex control flow, long methods, undesirable dependencies/coupling, etc. By contrast, the `StackAr` and `QueueAr` programs used in the experiments are very simple, and lack the use of common features, such as collections. Complexity and poor design make all software engineering tasks, including writing contracts, more difficult. Well-written, simpler code will cost less to work with.

Specification Complexity General-purpose libraries, such as the Java JDK, have extremely complicated specifications. In our experience, the JML contracts for client code are significantly less complicated. Just as it is not desirable to have low-skilled workers design libraries, it is not desirable to have low-skilled workers write specifications for libraries.

Additionally, in practice, JML specification writers simplify contracts by splitting them into cases. VeriWeb currently does not support cases, so users must use implication, which becomes unwieldy. One way to support cases in VeriWeb would be introduce a new subproblem type that asks the user to write (or select) the cases for a method contract.

8. Related Work

In this section, we survey related work in program verification and crowdsourcing.

Interfaces for Traditional Verification Tools The traditional program verification literature does not focus on user interface design, but sometimes contains it as a component. For example, Houdini, a static contract inference tool for ESC/Java, produces static HTML reports that contain hyperlinks to locations in the source code [18]. Flanagan and Leino reported that when working with Houdini, they repeatedly asked questions such as "Why didn't Houdini infer this precondition?" They note that anecdotal evidence shows that the presentation of the refuted annotations and the corresponding causes are the most important aspect of the user interface. Unfortunately, the authors did not enumerate any of their other questions.

Other work has focused on explicit deficiencies found in tools. For example, Kiniry [21] augmented the output of ESC/Java with warnings when the analysis may be either unsound or incomplete.

Pex for Fun (`http://www.pexforfun.com/`) is a web-based game designed to help students practice programming [34]. Levels come in two forms: puzzles and coding duels. Puzzles ask the user a question about a method, or group of methods, e.g., for what input values does the program throw an exception? In a coding duel, the user must write a method that behaves the same as a secret implementation. The Pex tool is used to generate relevant input values; problems can also include Code Contracts to guide Pex [3].

Usable Program Verification Microsoft Code Contracts [8] provides language support for integrating contracts into programs in the .NET languages, as opposed to using a separate specification language such as JML. The productized version of the tool can statically verify some properties; other specifications are checked at run time. Leino's Dafny language and verifier also aims to provide guarantees for imperative languages beyond that of extended static checking [23]. It provides support for a richer set of properties, translating Dafny code to the Boogie verification language [2] to produce the necessary conditions for a SAT solver.

There is a growing body of work on inferring contracts, using static analysis, dynamic analysis, and a combination of both. Independent of this work, Yi Wei et al. utilized traces to invalidate quantified expressions generated by generalizing contracts [38].

There is also a push to depart from the de facto delineation of program, contracts, and verifier — these new approaches often adopt a "pay-as-you-go" mentality. For example, pluggable type checking techniques [15, 29] allow users to incrementally extend the standard type system to check richer properties such as nullness, interning, and information tainting. In [32], Sheard et al. outline how the same philosophy can be applied in dependently typed languages that provide language-based verification. They argue that cognitive burden is reduced because properties fit naturally into the language.

Crowdsourcing Software Engineering Despite the growing base of crowdsourcing literature, there has been little academic exploration of crowdsourcing in software engineering. The only formal crowdsourcing research of which we are aware focused on end-user programming [33]. The commercial sector appears to have taken greater interest. For example, uTest is a start-up that uses crowdsourcing to provide on-demand software testing services [36].

Ad-hoc Labor Markets The creation of Mechanical Turk in 2005 spurred crowdsourcing research by providing easy access to a global workforce for ad-hoc tasks. As the field matures, research is transitioning from one-off proof of concept projects to crowdsourcing frameworks [25] and formal modeling of workflows [10].

As the understanding of dynamics of the labor market develops, tools that combine ad-hoc labor to achieve a larger goal are possible. For example, Soylent, a word processor backed by MTurk, introduces a Find-Fix-Verify pattern for managing worker variance in tasks [5]. In the Find phase, users identify parts of the document that need more work. In the Fix phase, workers propose revisions to the parts identified during the Find phase. Finally, in the Verify phase, workers determine which suggestions are the best.

Economics of Ad-Hoc Crowdsourcing Horton and Chilton [20] construct a formal model of labor supply based on reservation wage — the wage that causes the benefit of performing a task to exceed the benefits of performing other tasks. They characterize this wage with two experiments: (1) increasing task difficulty while keeping a workers wage constant and (2) decreasing the wage while keeping the difficulty constant. The task they used, clicking between two vertical rectangles, requires little skill. The lack of skill requirement is commonplace among the economic crowdsourcing literature.

Mason and Watts [26] explore the rationality of the Mechanical Turk workplace, that is to what extent the workspace satisfies traditional economic models in which pay determine economic quality and quantity. To vary the price paid to workers, they set a base rate for the HIT and use Mechanical Turk's preview functionality to inform workers of an additional bonus. They find that pay has a positive relationship with the quantity of work performed on a task, but does not have a measurable effect on the quality. Furthermore, they provide evidence that enjoyment, intrinsic motivation, and other factors play a significant role in the marketplace.

9. Conclusion

We have presented VeriWeb, a web-based tool for program verification that incorporates novel interface features to accommodate task decomposition and reduce the level of skill required to write verifiable specifications for programs. It is designed around the principles of active guidance and explanations in context, helping its users to do the right thing and to understand why it is the right thing.

To evaluate VeriWeb, we performed three experiments with two small subject programs. While we found evidence that current ad-hoc crowdsourcing marketplaces cannot support VeriWeb, we found that VeriWeb lowered the monetary and time cost of verification when using contract workers. Additionally, we observed that VeriWeb can be used collaboratively with minimal communication overhead.

Our vision is a world in which software verification is more economically feasible, and therefore is performed more often. A path to this vision is crowdsourcing: making verification tasks accessible to a broad range of developers, even those with relatively little training. Our work is one small step along this path.

Acknowledgments

We thank Colin Gordon, Brian Burg, and Sai Zhang for their feedback on VeriWeb's user interface. Additionally, we thank all of the students and workers who participated in pilot studies and provided invaluable feedback, and the anonymous reviewers for their thoughtful suggestions.

This material is based upon work supported by a National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124 and DARPA under Grant No. FA8750-11-2-0221.

References

[1] Amazon Mechanical Turk, Apr. 2012. `https://www.mturk.com/`.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. J. 002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[3] M. Barnett, M. Fahndrich, P. de Halleux, F. Logozzo, and N. Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *31st International Conference on Software Engineering*, pages 401–402, May 2009.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, pages 49–69, Mar. 2004.

[5] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 313–322, New York, NY, USA, 2010. ACM.

[6] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[7] B. V. Chess. Improving computer security using Extended Static Checking. In *IEEE Symposium on Security and Privacy*, pages 160–173, Berkeley, California, May 12–15, 2002.

[8] Code Contracts user manual. `http://download.microsoft.com/download/C/2/7/C2715F76-F56C-4D37-9231-EF%8076B7EC13/userdoc.pdf`, Sept. 2010.

[9] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS 2004, Revised Selected Papers*, volume 3362 of *LNCS*, pages 108–128, Marseille, France, Mar. 10–13, 2004.

[10] P. Dai, Mausam, and D. S. Weld. Decision-theoretic control of crowd-sourced workflows. In *AAAI Conference on Artificial Intelligence*, pages 1168–1174, 2010.

[11] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, Apr. 2008.

[12] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, July 23, 2003.

[13] D. L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, Jan. 1996.

[14] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, Dec. 18, 1998.

[15] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE*, pages 681–690, May 2011.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Programming*, 69(1–3):35–45, Dec. 2007.

[18] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, pages 500–517, Mar. 2001.

[19] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, June 2002.

[20] J. J. Horton and L. B. Chilton. The labor economics of paid crowdsourcing. In *Proceedings of the 11th ACM Conference on Electronic Commerce*, EC '10, pages 209–218, 2010.

[21] J. R. Kiniry, A. E. Morkan, and B. Denby. Soundness and completeness warnings in ESC/Java2. In *Proceedings of the Fifth International Workshop on Specification and Verification of Component Based Systems (SAVCBS)*, 2006.

[22] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM Softw. Eng. Notes*, 31(3), Mar. 2006.

[23] K. Leino. Dafny: An automatic program verifier for functional correctness. In E. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer Berlin / Heidelberg, 2010.

[24] K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In *Compiler Construction '98*, pages 302–305, Apr. 1998.

[25] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09. ACM, 2009.

[26] W. Mason and D. J. Watts. Financial incentives and the "performance of crowds". In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 77–85, New York, NY, USA, 2009. ACM.

[27] G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Palo Alto, CA, 1980. Also published as Xerox Palo Alto Research Center Research Report CSL-81-10.

[28] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE*, pages 11–20, Nov. 2002.

[29] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[30] Purchasing power parities for GDP and related indicators. `http://stats.oecd.org/Index.aspx?DataSetCode=PPPGDP`, Apr. 2012.

[31] T. W. Schiller and M. D. Ernst. Rethinking the economics of software engineering. In *FoSER*, pages 325–330, Nov. 2010.

[32] T. Sheard, A. Stump, and S. Weirich. Language-based verification will change the world. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 343–348, New York, NY, USA, 2010. ACM.

[33] K. T. Stolee and S. Elbaum. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 35:1–35:4, New York, NY, USA, 2010. ACM.

[34] N. Tillmann, J. de Halleux, and T. Xie. Pex for fun: Engineering an automated testing tool for serious games in computer science. Technical Report MSR-TR-2011-41, Microsoft Research, Redmond, WA, March 2011.

[35] U.S. Bureau of Labor Statistics. Computer software engineers, applications, May 2010. `http://www.bls.gov/oes/current/oes151031.htm`.

[36] uTest: Software testing, Apr. 2012. `http://www.utest.com`.

[37] vWorker.com: More capable, accountable and affordable. guaranteed., Apr. 2012. `http://www.vworker.com`.

[38] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proceedings of 2011 International Conference on Software Engineering (ICSE 2011)*, 2011.

[39] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley Longman, 1999.

Appendix: Edit Distance Adjustment

The edit distance metric assumes that each contract is either present or not present. However, VeriWeb introduces a third "state" for a contract, because VeriWeb remembers contracts that are not currently in use but may be tried or presented to the user in the future. This state is not relevant to the Eclipse computation.

We adapt the edit distance metric as shown in Table 1 on the following page. Any user-written clause that is not in the target set counts toward the edit distance, as does any clause that is in the target set but not in the candidate set either as a user-written or VeriWeb-suggested clause. In general, a VeriWeb-suggested clause is treated like a user-written one. The exception is that a correct postcondition that has not yet been proved by VeriWeb is not counted against the user. Such a clause typically exists only because the user has not proceeded to the appropriate postcondition subproblem; when the user does, the postcondition will be automatically added by VeriWeb without any human intervention.

A clause can be proven but "wrong" either because the proof depends on other wrong clauses, or because the target specification does not include that clause (another verifiable specification, however, might include the clause).

Sensitivity to Inferred Invariants The distance metric is sensitive to the inferred set of invariants. Let I be the inferred specification with preconditions I_R and postconditions I_E. Let X be the nearest verifiable specification with preconditions X_R and postconditions X_E. The distance for the Eclipse user is

$$|X \setminus I| + |I \setminus X|$$

, the number of conditions that were not inferred by Daikon plus the number of conditions that were incorrectly inferred by Daikon. For a possibly different nearest verifiable solution X, the distance for the VeriWeb user is

$$|X_R| + |X_E \setminus I_E|$$

, the number of preconditions in the solution plus the number of postconditions that inference failed to detect. The difference in distance is given by:

$$\begin{aligned}(|X_R \setminus I_R| + |X_E \setminus I_E| + |I \setminus X|) - (|X_R| + |X_E \setminus I_E|)\\ = (|X_R \setminus I_R| + |I_R \setminus X_R| + |I_E \setminus X_E|) - |X_R|\\ = (|I \setminus X| + |I_R \setminus X_R|) - (|I_R \setminus X_R| + |I_R \cap X_R|)\\ = |I \setminus X| - |I_R \cap X_R|\end{aligned}$$

, the number of incorrectly inferred conditions less the number of correctly inferred preconditions.

Invariant source	Eclipse	VeriWeb
Object invariants		
User-written	WRONG	n/a
Generalized	n/a	WRONG
Missing	RIGHT	RIGHT
Preconditions		
User-written	WRONG	WRONG
User-selected	n/a	WRONG
Not user-selected	n/a	RIGHT
Missing	RIGHT	RIGHT
Postconditions		
User-written	WRONG	WRONG
Proven	n/a	WRONG
Unproven	n/a	(none)
Missing	RIGHT	RIGHT

Table 1. Which clauses count toward the edit distance metric, when comparing a candidate specification (set of clauses) with a target specification. When considering a contract clause in the candidate set, "WRONG" means to count any clause that is not in the goal set, and "RIGHT" means to count any clause that is in the goal set.

The Eclipse column indicates that the edit distance for a precondition is the sum of the number of wrong user-written clauses, plus the number of missing (but necessary, or "RIGHT") clauses. The VeriWeb column indicates that the edit distance is the sum of the number of user-written and user-selected wrong clauses, plus the number of non-user-selected (but necessary) clauses, plus the number of missing (but necessary) clauses.