Toward Commoditized Verification

Todd Schiller
Michael Ernst
Verification: does the program fulfill the specified contract?

class Queue{
    ...

    /**
     * @requires x != null;
     * @ensures currentSize == \old(currentSize+1);
     * @exsures (QueueFullException) ...
     */

    public void enqueue(Object x)
        throws QueueFullException
    {
        ...
    }
}
Verifying a Specification

Source Code → Logical Formulas → Theorem Prover (SMT solver) → Counter-Example

∀ (a ∧ b) → ∃ (a ∧ b)

∀ (a ∧ b)

Specication
Verification isn’t Cost-Effective

- Evidence: only used for safety critical systems

- Essential complexity
  - Precision and completeness

- Accidental complexity
  - Tool design tradeoffs
  - Bad interface design

Labor intensive & Expert users
Worker Skill Spectrum

Verification Experts

Programmers

Engineers

Commoditization

Task-level Crowdsourcing
Barriers to Commoditization

• Interface usability is limited
  – Complicated internal representations

• Decomposition into subtasks is hard
  – Module and methods interdependent
  – Information loss
VeriWeb: a web IDE for writing verified specifications of existing code

- More cost- and time-effective than a traditional interface
- Enables *collaborative* verification via decomposition
VeriWeb Workflow

Skilled Developer

Writes Feature

Verifiable Specification

ESC/Java2
VeriWeb Interface

Contract Entry

Warnings & Specifications

Source Code
Instructions:
Drag condition fragments from the palette to the condition box to form conditions that MUST be true for the function to not throw an unexpected runtime exception. A submit button will appear when the condition in the box is complete.

Some fragments have yellow holes that must be filled with other fragments. To fill a hole, just drop a fragment onto it; you can later remove the fragment by clicking and dragging the fragment. NOTE: You can only fill fragment holes in the scratch pad and condition box.

You are done when there are no more errors detected; you can view errors by hovering your mouse over code that is underlined in red.

You can view a method or type's documentation by hovering your mouse over code that is underlined in blue.

Additionally, you can toggle the inline specifications for a method by clicking methods that are shown as buttons.

```java
/**
 * Construct the stack.
 * @param capacity the capacity.
 **/
public StackAr( int capacity )
{
    theArray = new Object[ capacity ];
topOfStack = -1;
    /*@ set theArray.owner = this; */
}
```
VeriWeb Outputs a Partial Specification

1. Client code does not throw unexpected exceptions

2. Properties (optionally) specified by the feature developer

3. Plus other necessary properties for #1 and #2
1. VeriWeb design principles
   – Active guidance
   – Explanations in context

2. Toward crowdsourcing: lessons learned

3. Challenges and open questions
Principle #1: Active Guidance

**Prevent mistakes**

Caveat: being *too* restrictive annoys users

\[ \forall i: \text{TopOfStack} \leq \text{theArray.length} - 1 \]

**Suggest actions**

\begin{itemize}
  \item \( \neg \text{result} \Rightarrow \text{this.currentSize} \geq 1 \)
  \item \( \neg \text{result} \Rightarrow \text{this.theArray[this.back]} \neq \text{null} \)
  \item \( \neg \text{result} \Rightarrow \text{this.theArray[this.front]} \neq \text{null} \)
  \item \( \text{result} \Rightarrow \text{this.currentSize} = 0 \)
  \item \( \text{this.currentSize} = 0 \Rightarrow \text{this.front} < \text{this.theArray.length} - 1 \)
\end{itemize}

Caveat: too many suggestions overwhelm users
Principle #2: Explanations in Context

Give *concrete* feedback about what the tool knows, and doesn’t know

Concrete Counter-Examples

<table>
<thead>
<tr>
<th>Name</th>
<th>Before Call</th>
<th>After Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>this.theArray</td>
<td>ref@6613606</td>
<td>ref@6613606</td>
</tr>
<tr>
<td>this.theArray[.]</td>
<td>length 2</td>
<td>length 2</td>
</tr>
</tbody>
</table>

Caveat: still must teach users how to use feedback

Contract Inlining

```java
public Object top() {
    // No Preconditions
    if (isEmpty()) {
        // Hide unproven postconditions
        POST: this.theArray != null
        POST: \result == true == (topOfStack == -1)
        POST: topOfStack <= this.theArray.length - 1
        POST: topOfStack >= -1
        POST: \forall i: (topOfStack + 1 <= i && i <= this.theArray.length - 1)
    }
}
```

Caveat: irrelevant feedback overwhelms users
Talk Outline

1. VeriWeb design principles
   – Active guidance
   – Explanations in context

2. Paying for verification: lessons learned

3. Challenges and open questions
Research Questions

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

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

3. How does decomposition and communication overhead affect the performance of collaborative verification?
What is the Cost of Verification?

• Hired programmers on vWorker
  – Workers bid hourly wage
  – Accepted 18 of 22 bids ($6 - $22 per hour)
  – No correlation between experience (skill) and wage

• Two treatments:
  – ESC/Java2 Eclipse Plugin (Control)
  – VeriWeb

• Learning effect control:
  – Tutorial writing a verified specification for a toy program
  – Comprehension quiz
Method Dependency Graph

Client

Stack ADT
VeriWeb Workers Finish Faster

Eclipse Plugin

VeriWeb

Distance to nearest solution vs. Elapsed Time (min.)
VeriWeb Workers Cost Less

Eclipse Plugin

VeriWeb

Distance to nearest solution

Money Spent ($)

Money Spent ($)
Counter-Examples Are Important

• All workers tried to introduce false properties

• Slowest Eclipse worker had most trouble

• Lifetime of false properties skewed:
  – Median: 2 min.
  – Mean lifetime: 34 min.
Can VeriWeb Use Crowdsourcing?

• Mechanical Turk: worker paid per small task

• Paid 15¢ - 30¢ per subproblem, determined randomly for each worker upfront

No. Low response and high reserve wage
Lessons and Challenges

• Additional compensation for learning to complete the tasks

• Chicken and egg problem: need many verification tasks to make learning attractive
Talk Outline

1. VeriWeb design principles
   – Active guidance
   – Explanations in context

2. Paying for verification: lessons learned

3. Challenges and open questions
Other Approaches

Approach

- UW: Players solve puzzles to infer qualified types
- Berkeley: Workers find visual patterns in traces for verification
- HKUST: “Players” chain together method calls for test generation

Must show benefit over automation of human strategy

Cannot not claim labor supply from small trials
Open Design and Research Questions

• What latency is acceptable?

• Is abstraction required to protect intellectual property?

• How do you control worker error?

Rethinking the Economics of Software Engineering (FoSER 2010)
VeriWeb: a web IDE for writing verified specifications of existing code

- More cost- and time-effective than a traditional interface
- Enables collaborative verification via decomposition