IDVE: an Integrated Development and Verification Environment for JavaScript

Christopher Schuster
University of California, Santa Cruz
cschuste@ucsc.edu

Cormac Flanagan
University of California, Santa Cruz
cormac@ucsc.edu

ABSTRACT

Program verifiers statically check programs based on source code annotations such as invariants, pre- and postconditions. These annotations can be more precise than simple types. For example, a sorting routine might be annotated with a postcondition stating that its result is sorted.

However, the verification process for these annotations can become complex. Therefore, simple error messages may not be sufficient to help the programmer resolve verification issues. In order to improve the programming experience for verified programming, this paper presents IDVE, an integrated development and verification environment that lets users interactively inspect and debug verification issues. The goal of IDVE is to provide a development tool that assists users with program verification analogous to how interactive step-by-step debugging avoids manual "printf debugging". IDVE enables programmers to interactively manipulate assumptions and assertions of verification conditions with a novel verification inspector, and IDVE automatically generates tests that serve as executable and debuggable counterexamples.

In addition to presenting the approach and implementation of the integrated development and verification environment, we also conducted a user study with 18 participants to evaluate how the proposed features of the environment are perceived. Participants with and without prior experience with program verifiers had to solve a series of simple programming and verification tasks and answer an online survey. Features of IDVE were generally seen as helpful or potentially helpful but user interface design is an essential factor for their utility.

CCS CONCEPTS

• **Software and its engineering** → *Integrated and visual development environments*; *Formal software verification.*

KEYWORDS

programming environments, program verification, JavaScript, test generation, interactive debugging

ACM Reference Format:
Christopher Schuster and Cormac Flanagan. 2019. IDVE: an Integrated Development and Verification Environment for JavaScript. In *Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19), April 1–4, 2019, Genova, Italy.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3328433.3328453

```
IDVE                                                        verify  run
Integrated Development and Verification Environment
 1  function abs(n) {                    [-1]
 2    requires(typeof n === 'number');
 3    ensures(res ⇒ res ⩾ 0); // does not hold
 4    if (n ⩾ 0) {
 5      return -n;            // due to a bug
 6    } else {
 7      return n;
 8    }
 9  }
10
11  const a = abs(-23);
12  assert(a ⩾ 0);
      verified assert: (a ⩾ 0)
```

Figure 1: The JavaScript function `abs` is annotated with pre- and postconditions. The assertion in line 12 can be statically verified but a bug in line 7 causes a verification error for the postcondition in line 3, so IDVE shows `-1` as counterexample for `n`.

1 INTRODUCTION

There are different ways to check whether a program is "correct", including dynamic testing and static type checking. Unfortunately, testing only checks a certain (finite) set of inputs and types may be too restrictive to express complex correctness properties. For example, correctness of a sorting routine requires that the output is both sorted and contains the same elements as the input. Program verification aims to prove such correctness properties for all possible inputs based on annotations such as pre-, postconditions, assertions and invariants.

Figure 1 illustrates the goals and the scope of the proposed programming environment with a concrete example. Here, **`requires`** and **`ensures`** are pseudo-functions calls that will be skipped during execution but are used to specify pre- and postconditions as a standard JavaScript boolean expression. Due to a bug, the `abs` function returns its argument as a negative number, violating the postcondition in line 3.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Programming '19, April 1–4, 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6257-3/19/04...$15.00
https://doi.org/10.1145/3328433.3328453

The prototype implementation of the integrated development and verification environment, abbreviated as IDVE, helps the programmer identify verification conditions and inspect potential verification errors. Figure 1 does not show the full programming environment, but it illustrates how symbols next to the line numbers are used to indicate verification conditions. Hovering over these marks with the mouse cursor display additional details – similar to type errors. For failed verification conditions, IDVE also displays counterexample values as editor popups. For example, it displays `-1` as a value for the function argument `n` that causes a violation of the postcondition.

Additionally, IDVE also enables programmers to inspect specific verification conditions by opening an interactive inspector panel (not shown in Figure 1) that lets users inspect, add and remove assumptions and assertions – similar to "watch expressions" in an interactive debugger. Thereby, the verification inspector allows programmers to explore the verifier state without manually adding **`assert`** statements to the code, analogous to how interactive debuggers let programmers avoid `printf` debugging. Finally, the environment also includes an integrated debugger for the automatically generated test cases that lists variables in scope, shows the current call stack and allows step-by-step debugging.

IDVE, the integrated development and verification environment presented in this paper is an extension to ESVERIFY [35], a program verifier for dynamically-typed JavaScript programs. JavaScript supports both object-oriented and functional programming but ESVERIFY focusses mostly on functional programs with higher-order functions and dynamic idioms and code styles such as polymorphic functions that behave differently based on the number and types of their arguments. The source code of ESVERIFY as well as a live demo are available are publicly available[1]

An essential part of the proposed environment is automatic generation of counterexamples for verification errors. Automatic generation of test cases is a common technique for program analysis, often used in combination with symbolic execution [41]. As counterexample for a failed verification condition, the test should serve as a concrete witness for an assertion violation but it also needs to be faithful to the original source code. For cases involving loop invariants and recursion, these two goals can come into conflict and different approaches offer different trade-offs.

Test generation involves runtime checking of function pre- and postconditions similar to dynamically enforced contracts [16]. Moreover, for programs with higher-order functions, automatic test generation also involves synthesis of function arguments [40, 46]. The function synthesis implemented for our test generator is based on mapping simple arguments values to return values and therefore limited to pure functions that do not manipulate objects. Finally, when a generated test serves as counterexample for a function specification, simply wrapping a function with a contract is not sufficient to cause an assertion violation, as the test generation also needs to generate a call to the wrapped function.

Finally, the environment and its usability for developing verified programs was evaluated with a user study with 18 participants that have at least basic knowledge of JavaScript. The test subjects were given a brief introduction to the features of IDVE, had to solve a series of simple programming tasks with the environment[2], and answered a brief survey about their experience[3]. Results indicate that more than half of the participants were able to use the features of IDVE effectively to solve the programming and verification tasks. All participants reported that they found the tools either helpful or potentially helpful. However, an improved user interface design might enable more programmers to successfully use these features.

To summarize, the main contributions of this paper are

(1) an extension for the ESVERIFY program verifier that automatically generates executable counterexamples as test cases for failed verification conditions with synthesis of function values and assertion-violating calls,
(2) the design and implementation of an integrated development and verification environment (IDVE) with a novel interactive verification inspector and debugging interface, and
(3) a user study to evaluate whether and how IDVE assists with simple programming and verification tasks.

The structure of the rest of the paper is as follows: Section 2 gives an overview of verification with ESVERIFY and discusses the relevant features of the integrated development and verification environment, Section 3 describes the design and implementation of the automatic test generation procedure, Section 4 describes the method and results of the user study, Section 5 discusses related work, and finally Section 6 concludes the paper.

2 OVERVIEW OF VERIFICATION WITH ESVERIFY AND IDVE

Our integrated development and verification environment, IDVE, targets a subset of ECMAScript/JavaScript. While we want to support dynamic idioms, higher-order functions and mutable variables, advanced object-oriented programming with prototypes as well as scripting language features such as metaprogramming and reflection are orthogonal to the proposed approach discussed in this paper.

IDVE is based on the ESVERIFY program verifier. A detailed discussion of the design and implementation of ESVERIFY is beyond the scope of this paper, but this section introduces its basic usage as well as its integration with IDVE.

2.1 Verification with ESVERIFY

ESVERIFY relies on source code annotations such as pre- and postconditions and invariants, written as *pseudo function calls* that will be skipped during execution. Here, the logical propositions of these annotations are pure boolean expressions embedded in JavaScript. Figure 1 shows an example of an annotated JavaScript program. Due to a bug, the `abs` function returns its argument as a negative number, violating the postcondition in line 3.

Essentially, ESVERIFY traverses the source code and generates verification conditions using a strongest postcondition predicate transformer approach [17, 32]. These verification conditions are then checked by SMT solving with z3 [15] or CVC4 [5, 6]. If the SMT

[1] Source code: https://github.com/levjj/esverify Live demo: https://esverify.org/try

[2] The tutorial steps as well as the experiments are listed in Appendix A and an archived version of the user study is available online at https://esverify.org/userstudy-archived.

[3] Survey results are included in Appendix B.

```
1  function inc (x) {
2    requires(Number.isInteger(x));
3    ensures(y => Number.isInteger(y) && y>x);
4    return x + 1;
5  }
6  function twice (f) {
7    requires(spec(f, (x) => Number.isInteger(x),
8                     (x,y) => Number.isInteger(y) && y > x))
9    ensures(g => spec(g,(x) => Number.isInteger(x),
10                       (x,y) => Number.isInteger(y) && y > 0))
11   return function (x) {            // should be y > x ^^^^^
12     requires(Number.isInteger(x));
13     ensures(y => Number.isInteger(y) && y > x);
14     return f(f(null)); // should be f(f(x))
15   };
16 }
17 const incTwice = twice(inc);
18 const y = incTwice(3);
19 assert(y > 3);
```

Listing 1: Verification example with a higher-order function `twice`. The pre- and postcondition of its function argument `f` and of the returned function `g` are both described with the `spec` syntax. Bugs in lines 10 and 14 cause verification errors.

solver cannot refute a verification condition, it is proven correct for all inputs; otherwise a potential counterexample is returned.

It is important to note that ESVERIFY does not infer invariants, so programs with loops or recursion require explicitly specified invariants and may fail to verify despite being correct.

2.2 Higher-order Functions

In order to support verification of higher-order functions, ESVERIFY introduces a special `spec` syntax to describe the minimum pre- and postcondition of a function value.

As an example, Figure 1 shows a function `twice` that expects a function argument `f`. Here, any integer value for `x` needs to satisfy the precondition of `f` and any result `y` returned by `f` needs to be an integer greater than the argument `x`. The return value of `twice` is itself a function that applies `f` twice. However, due to an error in line 10, the postcondition of `twice` cannot be verified and, additionally, a bug in line 14 violates the precondition of `f`. In Section 3.3, we will describe how IDVE automatically generates executable test cases that serve as counterexamples for these two errors.

2.3 Interactive Tool Support for Verification

In order to be useful in practice, program verification has to be integrated into the development process. Ideally, feedback provided by the verifier should be instantaneous, continuous, informative, comprehensible and actionable. Instantaneous feedback requires the verification procedure to be fast enough to avert noticeable delays. This also enables continuous feedback by implicitly invoking the verifier after each code change. Most program verifiers, including ESVERIFY, can check small to medium source files in less than a second and thereby enable sufficiently fast feedback. Providing comprehensible and actionable feedback, in contrast, is still a major challenge for program verification because the complexity of

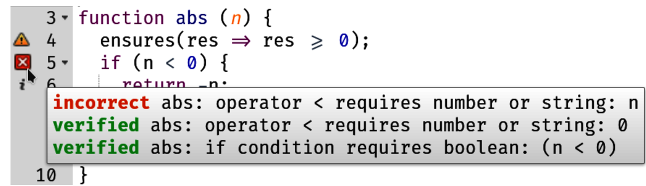

Figure 2: Verification conditions displayed as line markers with short error messages displayed as tooltips. Due to a missing precondition, the value of `n` may be incompatible with the `+` operator.

the verification procedure can result in errors that are hard to understand. IDVE is an integrated development and verification environment that integrates an interactive verification inspector and a counterexample debugger to address this issue. The implementation of IDVE is open source[4] and a live demo is available online[5].

2.4 Basic Line Markers

Each verification condition identified by the verifier corresponds to a location in the source code, such as a postcondition of a function definition, a function call that has to satisfy a precondition, or the arguments of a binary operator which have to adhere to a certain type.

As a basic form of environment integration, the source location of a verification condition can be displayed as an annotation or line marker in the code editor. Figure 2 shows an example of an editor with line markers to indicate verification conditions, using different icons for verified, unverified and incorrect results. Here, a result is considered *incorrect* if the generated test causes an assertion violation while an *unverified* result might have failed verification due to a missing loop invariant rather than an actual bug in the code. In addition to the icon, these line markers also include a short message that can be displayed by hovering the mouse cursor over the icon.

In addition to verification errors, this type of feedback is also common for type errors. Indeed, the line markers in IDVE are also used for other errors such as parsing and scoping issues.

While this integration is relatively simple, non-intrusive and self-explanatory, the provided feedback is limited and may not be sufficiently detailed to help the programmer understand and fix potential verification issues.

2.5 Verification Condition Inspector

By selecting one of the line markers discussed in the previous section, an additional panel can be opened with an interactive "inspector" for verification conditions.

Figure 3 shows an example with an active verification inspector. Here, the verification condition in line 4 cannot be verified and is selected in the editor on the left. The panel on the right then allows interactive inspection of the verification condition.

[4]Implementation source code: https://github.com/levjj/esverify-web
[5]Live demo of IDVE: https://esverify.org/idve

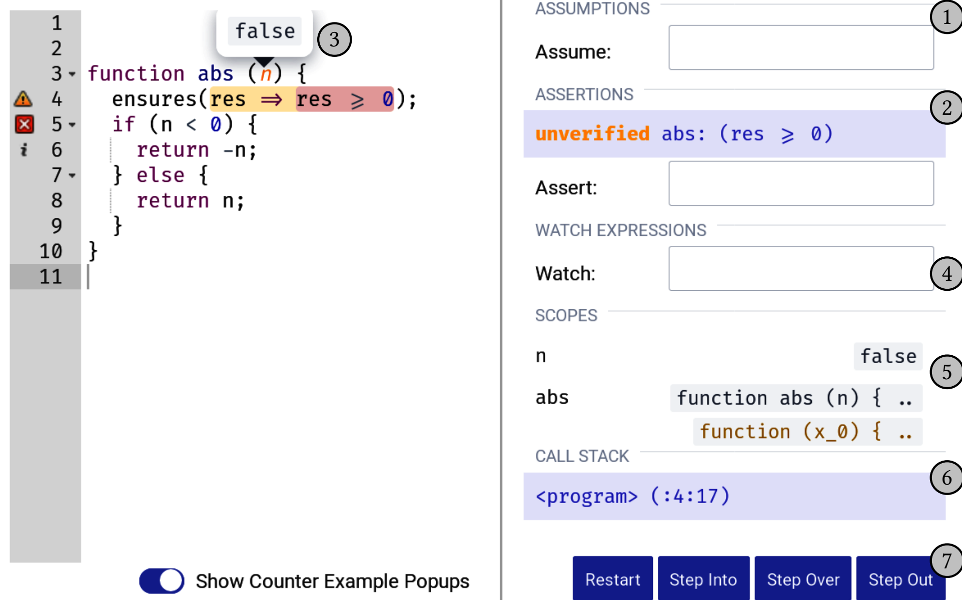

Figure 3: Selecting the unverified verification condition in line 4 opens a verification inspector on the right, showing assumptions, assertions and a debugger for the counterexample.

In particular, the inspector lists assumptions ① such as preconditions and invariants. While there are no relevant assumptions in this example, the user can enter an assumption in the form of boolean expressions and add it to the verification context for the selected verification condition. For example, by entering the JavaScript expression `typeof n === 'number'`, the verification condition will be re-examined with the new assumption, causing the verification of `res >= 0` to succeed.

Similarly, the inspector displays the asserted proposition ② but also allows additional assertions to be entered by the user and tested for the same assumptions and context. This feature of the verification inspector can be useful for interactive exploration and experimentation with the verification context without having to change the original source code (analogous to how interactive debuggers supersede "printf debugging"). This feature is novel in verified programming environments as existing environments such as Dafny IDE only display information about verification conditions without providing ways to interactively alter assumptions and assertions with a verification inspector.

2.6 Counterexample Popups

For each failed verification condition, a counterexample is synthesized based on the SMT solver output. This counterexample includes concrete JavaScript values for free variables such as function arguments and mutable variables in the surrounding scope. IDVE displays these counterexample values for the currently selected verification condition as popups ③ in the editor. These popups are directly connected to the relevant variable or parameter definition but they also obscure the source code below and they only display short summaries that may be inadequate for complex values such as nested objects and arrays. Visualizations of nested data structures is also an important challenge for regular debuggers and development tools [21].

2.7 Debugger Integration

Even with information about values of free variables such as those displayed with editor popups, it may not be obvious why a postcondition may not hold — especially for longer functions and methods.

In these cases, it might be helpful to inspect the current values of variables at different points in the function body. In general, this can be achieved by using an interactive step-by-step debugger. In the context of a failed verification condition, a counterexample test can be automatically generated for the purpose of debugging (see Section 3). Running this test will either result in an assertion violation, which serves as a concrete witness for a bug in the code or annotations, or a successful test execution without error, which indicates that a false positive was caused by the conservative static analysis of loops and recursion. In both cases, stepping through the code can help with understanding the verification issue and locating the root cause of the bug.

There are two possible approaches for debugging the counterexample test. On the one hand, the generated test could be "exported" and debugged with a traditional debugger in a separate tab or window. This would ensure a traditional debugging experience and could even enable the generated test to be added to an existing unit test suite. However, this approach requires the user to switch contexts between the original code and the test. Moreover, this approach exposes the automatically generated test code which may not be human readable and whose mapping to the original source code may not be obvious.

On the other hand, it is possible to hide the automatically generated test code and debug the counterexample directly on the level

of the original source program. Essentially, the debugger internally steps through the generated test while highlighting expressions and statements in the original source code that correspond to the current code fragment in the test. This approach avoids context switches but it might cause an unexpected order of execution steps in the visible source code if the control flow in the generated test differs from the control flow in the original program due to inserted dynamic checks or contract wrappers. Alternatively, the user could also disable stepping through user-written wrappers in order to avoid these jumps.

Figure 3 shows an integration using the latter approach. Here, the debugger is halted at the assertion `res >= 0`, and IDVE shows watched expressions (4), variables in scope (5), and the call stack (6). The user can add additional watch expressions and display their evaluation results. For example, entering `res` in (4) will show **`false`** as the returned value for this counterexample. Additionally, the execution can be stepped using standard debugger controls (7).

In contrast to debugging regular executions, the integrated debugger of the verification environment includes both a dynamic context from the actual test execution as well as a static context representing the verifier state at different points in the code. For example, for `abs`, the scope panel (5) contains both the function value used by the test execution in black and a synthesized function value in brown below. Only a single value is displayed if the value from the dynamic and static contexts agree. While this information is not relevant in this example, comparing the dynamic and static value of a mutable variable after a loop may help to understand missing or incorrect loop invariants (see Section 3.2).

Internally, the debugger is implemented as an interpreter that operates directly on the JavaScript AST of the test code. This incurs a high performance penalty over techniques such as source-to-source compilation [7] but it provides greater flexibility for incorporating additional features. For example, the IDVE debugger maintains both a dynamic execution and a static verifier context and it provides a better debugging experience for stepping through function wrappers.

3 AUTOMATIC TEST GENERATION FOR COUNTEREXAMPLES

This section describes the design and implementation of the verification process and particularly the automatic test generation[6].

3.1 Verification Process

Figure 4 illustrates the basic verification process including program verification and automatic test generation.

In summary, the **verification** step traverses the entire source program. At each statement and expression, the current verification context is used to generate verification conditions and augment the context for subsequent statements and expressions. Specifically, the verification context includes

- a logical proposition that acts as precondition,
- a set of free variables with unknown values, and
- a synthesized test with *holes*.

[6]The automatic test generator is already merged into ESVERIFY and its implementation is available at https://github.com/levjj/esverify/.

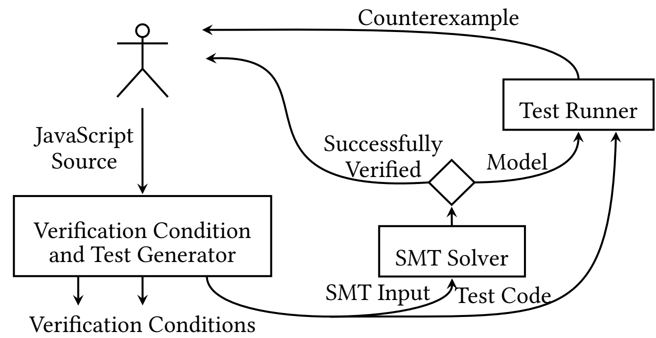

Figure 4: The basic verification workflow: ESVERIFY generates verification conditions to be checked by SMT solving. In order to better explain verification issues to the programmer, we extended ESVERIFY to also generate tests for failed verification conditions that serve as counterexamples.

Each returned verification conditions consists of such a context and an assertion to check, such as a function postcondition.

The next step of the verification process involves checking the verification condition with an **SMT solver**. If the solver cannot refute the proposition, the verification succeeded. Otherwise, the returned model includes an assignment of free variables that acts as a counterexample.

Such a model can then be combined with the synthesized partial test. Inserting concrete values into the holes of the test yields an executable counterexample that can be evaluated by a **test runner**. The test result provides useful feedback such as a dynamic assertion violation and enables inspection with interactive step-by-step debuggers and other tools.

3.2 Verification Errors and Assertion Violations

The purpose of the automatic test generation is to provide better feedback about failed verification conditions. To that end, the generated test should reflect both the specifics of the verification process as well as the actual behavior of the source code. In the case of loops and recursion, these two goals come into conflict because the static verifier overapproximates program behavior and thereby detects potential assertion violations that are not encountered by the actual evaluation of the program.

As an example, Listing 2 shows a program with an assertion that cannot be verified. For any statements below the **`while`** loop, the loop invariants can be assumed to hold and the loop condition will be false but, apart from these assumptions, all mutable variables in the code could have changed arbitrarily. Therefore, the assertion in line 11 cannot be verified despite `safe` remaining unchanged by the loop when executing the program. Adding `invariant(safe);` to the loop would let its verification succeed.

There are two possible options for generating a counterexample test in these situations. One option is to reuse large fragments of the original program for the test. If the test leads to an error or assertion violation, it can serve as a witness of an 'actual' error that

```
1  let safe = true;
2  let i = 0;
3  while (i < 3) {
4    invariant(Number.isInteger(i) && i <= 3);
5    if (i === 42) {
6        safe = false;
7    }
8    i++;
9  }
10 assert(i === 3); // verifiable
11 assert(safe);    // cannot be verified
```

Listing 2: Verifier detects assertion violation due to missing loop invariant.

```
1 let safe = true;
2 let i = 0;
3 // loop omitted; variables assigned according to SMT
4 safe = 0;
5 i = 3;
6 // statements after the while loop:
7 assert(i === 3);
8 assert(safe);
```

Listing 3: Replacing `while` loop in Listing 2 with counterexample values for test generation.

would also occur during normal execution and that can be fixed using the standard debugging process. However, for the example shown in Listing 2, running such a test would not result in an assertion violation because the original program execution satisfies the assertion in line 11. This indicates that the static analysis did not accurately model the actual program behavior. Instead, the verification error is caused by an insufficiently strong loop invariant or precondition/postcondition. Unfortunately, there is no feedback about which annotation might be missing or what the internal verification state is regarding mutable variables after the loop.

As a second option, the generated test for the failed assertion in Listing 2 can omit the original `while` loop and instead insert assignments to mutable variables according to the values in the SMT model. Listing 3 shows an example of such a test. Clearly, the generated test reliably causes an assertion violation. Also, by using values from the SMT model for mutable variables after the loop, the generated test might help the programmer better understand the verification process and its shortcomings, and how the loop invariants can be improved to satisfy the assertions below the loop. In this case, the loop invariants constrain the mutable variable `i` ↪ but the possibles values of `safe` are left unrestricted, so the SMT model may assign it `false` or even `0` after the loop. This is theoretically consistent with the specified invariants but different from the actual program behavior of the loop.

As outlined in this section, both the test case with the original source code as well as the model-based generated test case yield useful feedback to the programmer, serving the two competing goals of inspecting both the actual program behavior and static verification process.

To provide the benefits of both approaches, the esverify test generator retains the original loops but also enable programmers to query the variables in the verifier state with an integrated debugger.

```
1  function twice (f) {
2    f = function (x) { // spec(f, ...) in line 7
3      const y = f(x);
4      assert(Number.isInteger(y) && y > x);
5      return y;
6    };
7    const g = function (x) { // inner function in lines 11-15
8      assert(Number.isInteger(x)); // need to check precond.
9      const y = f(f(null));        // but can assume postcond.
10     return y;
11   };
12   g = function (x) { // spec(g, ...) in line 9
13     assert(Number.isInteger(x));
14     const y = g(x);
15     return y;
16   };
17   return g; // return the wrapped inner function
18 }
```

Listing 4: Transformed code for the `twice` function in Listing 1. The assignments in lines 2 and 12 install wrappers according to the `spec` in lines 7 and 9 of Listing 1.

3.3 Dynamic Checking of Assertions

The generated test for a verification condition consists of a transformed fragment of the relevant source code and a dynamically-checked assertion.

Assertions such as pre-, postconditions and invariants are specified as JavaScript boolean expressions. Therefore, dynamically checking these assertions as part of a test can be performed by evaluating these boolean expressions and throwing an exception if the evaluation result is different from `true`. If the source programs use exceptions handling, a potential assertion violation should be reported even if the exception is caught. However, esverify does not currently support exception handling and rejects programs with `try`/`catch` blocks.

In contrast to simple boolean expressions, function specifications using the `spec` syntax, as used by the `twice` function in Listing 1, cannot be checked dynamically for all values at the point of the assertion. Instead, the function argument is wrapped in a *contract* that enforces the specified pre- and postcondition for each subsequent call in the scope of this `spec`.

Listing 4 illustrates how the `twice` function can be transformed to enable dynamic checking of function specifications. The code shown here is slightly simplified. In particular, it does not collapse wrappers to avoid repeated and redundant wrapping. It is important to note that assuming and asserting a function specification result in a different transformation. In this example, the `twice` function and its returned inner function are assumed to adhere to their function specifications but the function argument `f` is not trusted to behave correctly when invoked with correct arguments. Therefore, the postcondition of `f` is dynamically checked but its precondition assumed, and conversely, the preconditions of `g` and

```
1 class A {
2   constructor (x) {
3     this.x = x;
4   }
5   m () {
6     assert(this.x > 0);
7   }
8 }
```

Listing 5: A simple class definition. The assertion in line 6 does not hold for all instances of `A`.

```
1 const this_0 = new A(false);
2 assert(this_0.x > 0);
```

Listing 6: Generated test for the failed assertion in line 6 of Listing 5.

the `spec` in the postcondition of `twice` are enforced but their postconditions are not. Incidentally, this mechanism is similar to blame assignment [1, 26].

The example in Listing 1 only includes first and second-order functions but it is also possible for a function specification with `spec` to occur within another `spec`. In that case, the transformation of function specifications is applied recursively, i.e. the inner wrapping code gets executed as part of the dynamic checks of the outer wrapper.

This technique of dynamically checking assertions is only used in generated counterexample test but it could also be adapted to support sound execution of partially verified programs, similar to "soft verification" [33] and "gradual verification" [4].

3.4 Synthesis of Counterexample Values

As shown in Figure 4, if the SMT solver refutes a verification condition, it returns a model that assigns values to free variables in the verification condition. In order to generate executable tests, these values have to translated from an SMT internal format to valid JavaScript expressions that can be inserted into the testing code.

For opaque JavaScript values such as `undefined`, **`null`**, **`true`** and **`false`** this translation is trivial. Numeric values in JavaScript conflate integers and floating point numbers and therefore are represented as either integers or real numbers in the SMT format in order to support both integer semantics for array indexing as well as floating point semantics for arithmetic operators[7]. Modern SMT solvers such as z3 [15] and CVC4 [5, 6] contain theories for strings with support for indexing and substrings[8]. Therefore, these JavaScript strings can be directly represented as SMT strings.

esverify provides limited support for object-oriented programming by means of immutable "classes" without inheritance and only trivial constructors. As an example, Listing 5 shows a class definition with a method `m` containing an incorrect assertion. Adding

[7]The SMTLIB standard also includes floating point values which, in contrast to real numbers, have limited precision. Unfortunately, current SMT solver implement these as bitvectors which negatively affects solving performance.

[8]Theories for strings have been added relatively recently and are still prone to errors such as SMT solver timeouts when converting strings with `str.to.int`.

```
1  let f = function (x) {          // synthesized function as
2    if (x === 3) {               // part of counterexample
3      return 9174;
4    }
5    return false;
6  };
7  f = function (x) {             // spec(f, ...) in line 7
8    assert(Number.isInteger(x)); // need to check precond.
9    return f(x);                 // but can assume postcond.
10 };
11 let x = 3;                     // variable in scope (unused)
12 f(null);
```

Listing 7: Generated test for precondition of `f(null)` in line 9 of Listing 1.

a sufficiently strong class invariant to `A` would let verification succeed. In the generated test shown in Listing 6, **`this`** is renamed and initialized with a constructor invocation according to the SMT model.

For plain JavaScript objects/records and arrays, the test generation follows a similar strategy. Due to the modeling of data structures as immutable values instead of heap references, the generated counterexamples deviate from standard JavaScript semantics with regards to aliasing. Similarly, counterexamples with cyclic references in data structures are not currently supported.

In order to generate counterexamples for higher-order functions, the test generator has to synthesize function values. Program synthesis is an active research topic with various different approaches and techniques [2, 19, 45] but esverify only supports a limited synthesis for the purpose of test generation. In particular, the synthesis of function values is limited to pure functions that map primitive argument values to return values. Thereby, the function can be expressed as a series of conditionals.

Listing 7 shows a generated test for the violated precondition of `f(null)` in line 9 of Listing 1. Here, a synthesized function value is assigned to `f` and then wrapped according to the specification in line 7 of Listing 1, resulting in an assertion violation when invoked with **`null`**. The synthesized function is based on a partial mapping and might include constants picked nondeterministically by the SMT solver. Therefore, the synthesized function may not adhere to the function specification when invoked with other arguments not included in the SMT mapping. This is why the synthesized function `f` returns a seemingly random number such as `9174` for the argument `3` but does not return "correct" values for other arguments.

3.5 Generating Counterexample Function Calls

When asserting function specifications, the specification is transformed to a wrapper in the generated test. However, without subsequent calls, the test would simply end without triggering an assertion violation. Therefore, the test generator also needs to synthesize a violation-provoking call after installing the wrapper for the asserted specification.

As an example, the postcondition of the `twice` function in Listing 1 does not hold because the inner postcondition in line 12 is not satisfied by the returned function. Listing 8 shows a simplified

```
1  let f = function (x) {   // synthesized function as
2    if (x === -2) {        // part of counterexample
3      return -1;
4    }
5    if (x === -1) {
6      return 0;
7    }
8    return false;
9  };
10 const g = function (n) { // original body of twice
11   return f(f(n)); // use f(f(n)) instead of f(f(null))
12 }
13 g = function (x) {       // spec(g, ... ) in line 12
14   const y = g(x);
15   assert(Number.isInteger(y) && y > 0);
16   return y;
17 };
18 g(-2);                   // synthesized function call
```

Listing 8: Generated test for the postcondition in line 9 of Listing 1. In addition to synthesizing `f` and wrapping the returned function `g`, it also generated a call.

generated test with a synthesized function value for `f`, the original function body of `twice` (assuming `f(f(null))` is replaced by `f(f(n))`), and a wrapper for the function specification in the postcondition of `twice`. Additionally, the generated test also includes a synthesized function call `g(-2)` that causes an assertion violation in line 15. The argument values for this violation-provoking function call are determined based on the SMT model.

4 EVALUATION AND USER STUDY

IDVE was evaluated with a user study with 18 participants. This section describes the relevant research questions, the design of the user study, its results, and potential threats to validity.

4.1 Research Questions

In contrast to the program verifier itself, the design of the integrated development and verification environment is difficult to evaluate due to the subjective experience of programmers and the large solution space. This user study aims to provide insight into answering the following three research questions in order to inform future designs of such environments.

RQ1: Can IDVE assist in the development process? Do programmers use features such as line markers, interactive manipulation of assumptions and assertions, counterexample editor popups and integrated debugging if these features are available for solving a given task?

RQ2: Is the proposed user interface helpful and intuitive? Careful consideration is required for the design of user interfaces of development environments in order to balance the amount of information and interactive controls. Therefore, this user study should determine whether the proposed design is generally perceived as intuitive or as overwhelming and cumbersome.

RQ3: How does programming proficiency and prior experience with program verification affect utility? The proposed environment should ideally be accessible and usable by both novices and experienced programmers. However, programming expertise and experience with program verification might be a prerequisite for effectively using the proposed features of the environment.

4.2 Methodology

IDVE is still an early prototype and not yet ready for productive software development. Therefore, the user study focuses on how proposed features can be used for smaller programming and verification tasks. Test subjects had no prior experience with IDVE itself but indicated to have at least basic knowledge of JavaScript and might have used other program verifiers before.

The user study was conducted entirely online. Subjects were recruited with an invitation sent to a public mailing list and remained anonymous. 18 adults entered the study and were presented with a brief introduction of IDVE, followed by a series of programming and verification tasks, and finally surveyed about their experience using the tool.

Appendix A lists instructions, provided code and hints for the tutorial and experiments. Additionally, an archived version of these tasks is available online at https://esverify.org/userstudy-archived.

For the first step, a guided tutorial introduced

- the source code editor itself,
- a simple verification example similar to the one in Figure 1,
- an interaction with the verification inspector as described in Section 2.5, and
- an interaction involving stepping through a generated test with the integrated debugger discussed in Section 2.7.

After the tutorial, participants solved three short tasks:

1. The first experiment involves an incorrect factorial function that causes an infinite recursion for negative arguments. This task can either be solved by changing the precondition or by changing the implementation of the function. Participants could use the verification inspector and the integrated counterexample debugger but editor popups were disabled.
2. For the second experiment, a correct implementation of six-sided dice rolling function was given. However, the function was missing postconditions necessary for verification of the subsequent code. Editor popups and the verification inspector were both disabled, so only basic line markers were available for this experiment.
3. The third and final experiment involved a function for converting the number of minutes since midnight into a 24-hour digital clock format. The provided code included bugs in both the annotations as well as the implementation. Both the verification inspector and editor popups were available but the integrated debugger was disabled.

All experiments could be skipped at any time and did not measure time or success. Instead, test subjects proceeded to the next experiment at their own discretion and filled out a survey form about their experience at the end.

4.3 Results

A full record of survey answers for all 18 participants including written comments can be found in Appendix B. These answers provided empirical evidence towards answering the research questions above.

RQ1: Can IDVE assist in the development process? According to the survey results shown in Table 1, the usage and the perceived benefits vary for the three main features of IDVE. While half of the participants made use of counterexample popups, the verification inspector was only used by 39 percent of participants, and the integrated debugger by 28 percent. In total, 15 participants reported using at least one of the tools. When features were used, they are generally seen as helpful or at least potentially helpful. This indicates that IDVE can effectively assist programmers.

RQ2: Is the proposed user interface helpful and intuitive? As shown in Table 1, the verification inspector, the counterexample editor popups and the integrated debugger were considered helpful by 33/55/44 percent of the subjects. Another 50/39/44 percent reported that these features could be helpful with an improved user interface. Additionally, a third of the subjects tried to use the verification inspector and the counterexample popups but reported being unsuccessful. Overall, the results suggest that the user interface is an important factor for using verification in practice.

Incidentally, half of the participants did not try to use the integrated debugger despite considering it helpful or potentially helpful. This might be a result of the experimental setup with programming tasks that were too trivial to require debugging but it might also indicate that the debugger integration might benefit the most from user interface improvements.

RQ3: How does programming proficiency and prior experience with program verification affect utility? As part of the user study, participants ranked their JavaScript proficiency on a scale from 1 (novice) to 5 (expert) and indicated whether they had prior experience with program verification. Table 2 shows how this related to usage and perceived benefits of IDVE. Here, it is most noteworthy that all participants, including those without JavaScript expertise or prior verification experience, found at least one of the the features helpful. Also, no significant differences were reported on the actual usage of these features in the experiments. While these results suggest that IDVE is accessible for both beginner and experienced programmers, the number of test subjects may be too low to fully support these conclusions. In fact, three of the participants remarked in written feedback that a more comprehensive tutorial about program verification would have been helpful.

4.4 Threats to Validity

The user study had a limited scope with only three short programming tasks and 18 participants. Therefore, it is possible that a broader user study with larger programming projects and more participants would yield different results. However, while scalability could be a concern for the runtime performance of the verifier, it can be expected that the responses of this user study regarding usability are at least indicative of a general trend that would also be observed by a larger user study.

Additionally, the results of this user study might be specific to JavaScript. It is possible that this approach for an integrated development and verification environment would be inadequate or unpractical for a different programming language or a different domain. For example, the integrated debugger for automatically generated tests may not be applicable to programs involving concurrency or input/output to external services or components.

5 RELATED WORK

The work presented in this paper comprises program verification, development environments and automatic test generation; each of which has been studied in prior research.

5.1 Program Verification

While this paper focuses on the verification environment, it is based on existing research in program verification [35]. In particular, the esverify program verifier is closely related to Dafny [29, 30], a verified programming language with static types, and LiquidHaskell [44], a dependent and refinement type system for Haskell. However, the verification system presented in this paper targets JavaScript, a dynamically-typed language. Another recent static verifier for JavaScript is JaVerT [18]. However, in contrast to esverify, JaVerT uses segmentation logic to reason about heap-allocated objects; but does not support higher-order functions.

5.2 Integrated Verification Tools

There has also been prior work on debugging tools in the context of program verification.

Even without static analysis, annotations such as function contracts may benefit from tool support such as dynamic activation/deactivation [24] and dedicating debugging features [3].

Tymchuk et al. recently investigated the integration of static analysis in a development environment by conducting interviews and concluded that this integration is essential for adoption [43].

For program analysis with symbolic execution, Hentschel et al. recently presented a Symbolic Execution Debugger that is integrated into Eclipse and offers a similar user experience to traditional debuggers [23].

In the context of program verification, work on dedicated debugging tools overlaps with research on interactive theorem provers. For example, the Proof Script Debugger for the KeY System [8] offers step-by-step debugging for proof scripts and inspection of the verifier state as assistance for theorem proving.

The most closely related research regarding integrated development and verification environments is the work on tools for Boogie, Dafny and Viper [39]. In particular, Le Goues et al. presented a verification debugger for Boogie, a low-level verification language [28] used internally by verifiers such as Dafny. At the same time, Dafny programs can also be debugged with an integrated development environment called Dafny IDE [10, 31] that addresses feedback about verification issues similar to IDVE. The Dafny IDE also allows inspection of counterexamples, including the state of variables at different stages in the verification of a function body. However, in contrast to IDVE, it does not enable the user to interactively modify assumptions and assertions of a verification condition.

Response (%)	Verification Environment Feature		
	Verification Inspector	Counterexample Popups	Integrated Debugger
Used this feature in experiments	39	50	28
Unsuccessfully tried using it	33	33	22
Did not use it	27	17	50
The feature is helpful	33	55	44
It could be helpful with different UI	50	39	44
It is not useful for development	6	6	6
It impairs the development process	11	0	6

Table 1: Participants indicated which features were used in the experiments and whether these features are seen as helpful.

Experience with	JavaScript					Verifiers	
	1	2	3	4	5	no	yes
# Participants	1	2	3	6	6	8	10
Successfully used features	1	2	3	5	4	6	9
Sees features as potentially helpful	1	2	3	6	6	8	10

Table 2: Usage and perception of verification environment features in relation to self-proclaimed proficiency.

5.3 Automatic Test Generation

While the goal of this paper is focused more on programming environment, Section 3 described the automatic test generation of verification counterexamples as a basis for further environment integration. Counterexample generation is an essential concept for both SMT solving and theorem proving [13]. Automatic test generation, i.e. generation of executable counterexamples, extends this idea and is a common technique for both program analysis and verification.

For program analysis, techniques such as symbolic execution, "whitebox testing" and parameterized testing [12, 20, 37, 41, 42] use path conditions to reason about control flow and generate random inputs in order to explore more program paths and thereby achieve higher test coverage. The approach in this paper, however, is based on formal verification rather than symbolic execution. Here, the main difference is the reasoning about loops and recursion. While verification requires manual annotations to avoid false positives, symbolic execution approximates the program behavior and may produce false negatives.

On a side note, This paper targets JavaScript, a dynamically-typed language, but automatic generation of error witnesses has also been explored for type errors in OCaml [36].

Heidegger and Thiemann previously presented a system for annotating JavaScript code with contracts that are used for guided random testing [22]. However, instead of using a static analysis such as program verification, the contracts have to be labelled explicitly by the programmer to guide the random testing.

Similarly, Klein et al. proposed a system in which test inputs are generated for stateful programs by randomly creating primitive values and objects, calling random functions and methods available in the current context and synthesizing function bodies [27]. This approach is similar to the test generation presented in Section 3. However, it provides better support for object-oriented programs (see also Thummalapenta et al. [40]) and it might not be able to explore deeper issues due to the black box generation of test inputs.

Automatic test generation has also been previously explored in the context of program verification in systems such as KeY [9], for techniques such as abstract interpretation [47], and for runtime verification [38].

For partially verified code, Christakis et al. proposed to use residual assumptions for complementary checking and dynamic testing in order to find more errors [11, 12].

Alternatively, conditional model checking is based on a condition that models control flow for properties that cannot be verified. This condition can either be translated back into a program that represents the unverified parts, or, alternatively, the unverified assertions can be used as slicing criterion for the original program. The residual program is then used for dynamic testing [14]. While conceptually similar, the automatic test generation presented in this paper targets higher-order functional programs and requires explicit invariants for loops.

More recently, Nguyen and Van Horn presented an approach for generating counterexamples for high-order functional programs. The paper uses a restrictive core language similar to simply-typed lambda calculus and finds counterexamples by SMT solving with an approximation relation for stateful programs [34].

Finally, for partially verified programs, verified assertions and invariants that depend on unverified verification conditions can be vulnerable, so robustness testing might be applicable [25].

6 CONCLUSIONS

Program verifiers enable expressing and checking various correctness properties but understanding and debugging resulting verification errors can be difficult. In this paper, we proposed both an integrated development and verification environment that automatically generates executable counterexamples for both first-order and higher-order functional programs.

These generated tests are used internally by IDVE, a prototype implementation of an integrated development and verification environment. It enables step-by-step debugging of these counterexamples and inspection of verification conditions including the option to interactively add or remove assumptions and assertions.

To evaluate this approach, we conducted a user study with 18 participants and conclude that the proposed development and verification environment can assist programmers and is generally seen as helpful, especially its feature for displaying counterexample values as editor popups. However, the user study also found that interface design is an important factor that could be improved to ensure that the proposed environment integration is useful in practice.

There are still open questions that could be addressed by future work such as whether this approach scales to larger applications with multiple developers and how it can be applied to other domains and programming paradigms.

REFERENCES

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *POPL '11*.
[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* (2015).
[3] Ryoya Arai, Shigeyuki Sato, and Hideya Iwasaki. 2016. A Debugger-Cooperative Higher-Order Contract System in Python. In *Programming Languages and Systems*.
[4] Stephan Arlt, Cindy Rubio-González, Philipp Rümmer, Martin Schäf, and Natarajan Shankar. 2014. The Gradual Verifier. In *NASA Formal Methods*.
[5] Clark Barrett and Sergey Berezin. 2004. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *CAV'04*.
[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV'11*.
[7] Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. [n. d.]. Putting in All the Stops: Execution Control for JavaScript. In *PLDI'18*.
[8] Bernhard Beckert, Sarah Grebing, and Alexander Weigl. 2018. Debugging Program Verification Proof Scripts (Tool Paper). *CoRR* (2018).
[9] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. 2007. *Verification of Object-oriented Software: The KeY Approach*.
[10] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. 2016. Integrated Environment for Diagnosing Verification Errors. In *TACAS'16*.
[11] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2012. Collaborative Verification and Testing with Explicit Assumptions. In *FM'12*.
[12] M. Christakis, P. Müller, and V. Wüstholz. 2016. Guiding Dynamic Symbolic Execution toward Unverified Program Executions. In *ICSE'16*.
[13] Simon Cruanes and Jasmin Blanchette. 2016. Extending Nunchaku to Dependent Type Theory. In *Hammers for Type Theories (HaTT 2016)*, Vol. 210. 3–12.
[14] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. 2015. Just Test What You Cannot Verify!. In *Fundamental Approaches to Software Engineering*.
[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08*.
[16] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *ICFP '02*.
[17] Cormac Flanagan and James B. Saxe. 2001. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *POPL '01*.
[18] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript verification toolchain. *POPL'18* (2018).
[19] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *ICSE 2014*.
[20] X. Ge, K. Taneja, T. Xie, and N. Tillmann. 2011. DyTa: dynamic symbolic execution guided with static verification results. In *ICSE'11*.
[21] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584.
[22] Phillip Heidegger and Peter Thiemann. 2010. Contract-Driven Testing of JavaScript Code. In *Objects, Models, Components, Patterns*.
[23] Martin Hentschel, Richard Bubel, and Reiner Hähnle. 2018. The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer* (2018).
[24] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. 2010. Dynamic Contract Layers. In *SAC '10*.
[25] Stefan Huster, Jonas Ströbele, Jürgen Ruf, Thomas Kropf, and Wolfgang Rosenstiel. 2017. Using Robustness Testing to Handle Incomplete Verification Results When Combining Verification and Testing Techniques. In *Testing Software and Systems*.
[26] Matthias Keil and Peter Thiemann. 2015. Blame Assignment for Higher-order Contracts with Intersection and Union. In *ICFP'15*.
[27] Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2010. Random Testing for Higher-order, Stateful Programs. In *OOPSLA '10*.
[28] Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. 2011. The Boogie Verification Debugger (Tool Paper). In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.).
[29] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR'10*.
[30] K. Rustan M. Leino. 2013. Developing Verified Programs with Dafny. In *ICSE'13*.
[31] K. Rustan M. Leino and Valentin Wüstholz. 2014. The Dafny Integrated Development Environment. In *Workshop on Formal Integrated Development Environment, F-IDE 2014*.
[32] C. Belo Lourenco, M. J. Frade, S. Nakajima, and J. Sousa Pinto. 2018. A Generalized Approach to Verification Condition Generation. In *COMPSAC'18*.
[33] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2017. Soft Contract Verification for Higher-order Stateful Programs. *POPL'17* (2017).
[34] Phuc C. Nguyen and David Van Horn. 2015. Relatively Complete Counterexamples for Higher-order Programs. In *PLDI '15*.
[35] Christopher Schuster, Sohum Banerjea, and Cormac Flanagan. 2018. esverify: Verifying Dynamically-Typed Higher-Order Functional Programs by SMT Solving. In *IFL '18*.
[36] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic Witnesses for Static Type Errors (or, Ill-typed Programs Usually Go Wrong). In *ICFP'16*.
[37] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750
[38] Ofer Strichman and Rachel Tzoref-Brill (Eds.). 2017. *Haifa Verification Conference, HVC 2017*.
[39] A. J. Summers and P. Müller. 2018. Automating Deductive Verification for Weak-Memory Programs. In *TACAS'2018*.
[40] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing Method Sequences for High-coverage Testing. In *OOPSLA '11*.
[41] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET. In *TAP'08*.
[42] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized Unit Tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 253–262.
[43] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2018. JIT Feedback—what Experienced Developers like about Static Analysis. *International Conference on Program Comprehension* (2018).
[44] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan Scott, Ryan Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement Reflection: Complete Verification with SMT. In *POPL'18*.
[45] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program Synthesis Using Abstraction Refinement. *POPL'18* (2018).
[46] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated Fixing of Programs with Contracts. In *ISSTA '10*.
[47] Greta Yorsh, Thomas Ball, and Mooly Sagiv. 2006. Testing, Abstraction, Theorem Proving: Better Together!. In *ISSTA '06*.

A USER STUDY TUTORIAL AND EXPERIMENTS

This appendix lists all tutorial steps and programming tasks that were part of the user study. Instructions, code and hints were displayed as a series of online web pages with a live web-based programming environment. An archived version of the user study including tutorial and programming tasks is available online at https://esverify.org/userstudy-archived.

A.1 Tutorial 1: JavaScript Live Editing

Edit and run a simple JavaScript program

Instructions

This user study involves interactions with a programming environment. The source code can be edited directly and the program can be executed in the browser. Test the editor by fixing the JavaScript program such that it computes the correct area of a rectangle.

Provided Code

```
1  // This is a live demo, simply edit the code and click run
2
3  const height = 3;
4  const width = 4;
5  const area_of_rect = height * height;
6
7  // should print '12', but prints '9' instead
8  alert(area_of_rect)
```

Steps and Hints

(1) Click the **run** button to see the result of the computation.
(2) Change the source code to compute the correct area of a rectangle.
(3) Click the **run** button again to test the code.

A.2 Tutorial 2: Program Verification With Pre- and Postconditions

Verify the given annotated `max` function and fix potential issues.

Instructions

esverify extends JavaScript with special syntax to annotate functions with pre- and postconditions. These are written as pseudo function calls that are skipped during evaluation. The following example includes an incorrect `max` function that should be fixed such that it returns the maximum of its arguments and verification succeeds.

Provided Code

```
1  // returns the maximum of the two provided numbers
2  function max(a, b) {
3    requires(typeof(a) === 'number');
4    requires(typeof(b) === 'number');
5    ensures(res => res >= a);
6    ensures(res => res >= b);// postcondition does not hold
7
8    if (a >= b) {
9      return a;
10   } else {
11     return a; // <- due to a bug in the implementation
12   }
13 }
```

Steps and Hints

(1) Click the **verify** button to verify all assertions in the code.
(2) The second postcondition does not hold due to a bug in the implementation.
(3) Change the source code to return the correct maximum of `a` and `b`.
(4) Click the **verify** button again to ensure that the new code verifies.

A.3 Tutorial 3: Interactive Verification Condition Inspector

Inspect a verification issue to understand and interactively explore assumptions and assertions.

Instructions

The following example includes a `max` function with missing preconditions. To better understand the problem, the esverify programming environment provides an interactive inspector for verification conditions that explains assumptions, assertions and counterexamples if available. This inspector also allows interactively adding assumptions and assertions.

Provided Code

```
1  // returns the maximum of the two provided numbers
2  function max(a, b) {
3    ensures(res => res >= a);
4    ensures(res => res >= b);
5
6    if (a >= b) {
7      return a;
8    } else {
9      return b;
10   }
11 }
```

Steps and Hints

(1) Click the **verify** button to verify all assertions in the code.
(2) Click on the yellow triangle in front of line 3 to select the verification condition.
(3) The panel on the right lists assumptions and assertions and the editor shows values for the counterexample as popup markers.
(4) According to the editor popups, the postcondition does not hold if the arguments are not numbers. Check this hypothesis by entering `typeof a === 'number'` next to 'Assume:' and confirm this with by pressing the enter/return key.
(5) Also add the assumption `typeof b === 'number'`.
(6) With these assumptions, the postcondition can be verified.

A.4 Tutorial 4: Verification and Debugger Integration

Query the counterexample and step through the code.

Instructions

For each unverified verification condition, the counterexample values can be used to execute the code with an interactive debugger. The debugger shows variables in scope, the current call stack and allows step-by-step debugging. Additionally, the debugger can be queried with watch expressions.

Provided Code

```
1  // returns the maximum of the two provided numbers
2  function max(a, b) {
3    requires(typeof(a) === 'number');
4    requires(typeof(b) === 'number');
5    ensures(res => res >= a);
6    ensures(res => res >= b);
7
8    if (a > b) {
9      return a;
10   }
11   if (b > a) {
12     return b;
13   }
14 }
```

Steps and Hints

(1) Click the **verify** button to verify all assertions in the code.
(2) Click on the first incorrect verification condition in line 5.
(3) The verification inspector in the panel on the right lists watch expressions, variables in scope and the call stack.
(4) In this case, the counterexample uses `0` for both `a` and `b`.
(5) To query the return value, enter `res` next to 'Watch:'.
(6) It seems the function returns `undefined`.
(7) To see the control flow, step through the code by clicking **Restart** and then clicking **Step Into** about eight times.
(8) It seems none of the two `if` statements returned a value when stepping through the code with this counterexample.

A.5 Experiment 1: Factorial

Instructions

This first experiment involves an incorrect factorial function. This example can either be fixed by adding a stronger precondition or by changing the `if` statement. You can use the verification inspector and the integrated counterexample debugger. Click 'Next' if you fixed the example or if you want to move to the next experiment.

Provided Code

```
1  // returns the factorial of the provided argument
2  function factorial(n) {
3    requires(Number.isInteger(n));
4    ensures(res => res >= 1);
5
6    if (n === 0) {
7      return 1;
8    } else {
9      return factorial(n - 1) * n;
10   }
11 }
```

Steps and Hints

No steps or hints.

A.6 Experiment 2: Dice Rolls

Instructions

This experiment involves an function for rolling a six-sided dice. A correct implementation is given and the following assertions should be verifiable but the postconditions are missing. The verification inspector is not available. Click 'Next' if you fixed the example or if you want to move to the next experiment.

Provided Code

```
1  // Roll a six-sided dice
2  function rollDice () {
3    // missing annotations
4    // ensures(res => ...);
5    return Math.trunc(Math.random() * 6) + 1;
6  }
7
8  const r = rollDice() + rollDice() + rollDice();
9  assert(r >= 3);
10 assert(r <= 18);
```

Steps and Hints

(1) Add missing postconditions with `ensures(res => ...);` in order to verify the assertions.
(2) You can verify code but there is no verification inspector.
(3) Click 'Next' if you fixed the example or if you want to move to the next experiment.

A.7 Experiment 3: Digital 24 Hour Clock

Instructions

This is the third and final experiment of this user study. Given the number of minutes since midnight, you should return time in a 24-hour digital clock format. You need to add an additional precondition and change the returned value. (Hint: `Math.trunc` rounds a number down to an integer.) You can use the verification inspector and the editor counterexample popups. Click 'Next' if you fixed the example or if you want to finish the experiments.

Provided Code

```
1  // Given the number of minutes since midnight,
2  // returns the current hour and minute as object
3  // in a { h: 0-23, m: 0-59 } format
4  function clock_24 (min) {
5    requires(Number.isInteger(min) && 0 <= min);
6    ensures(res => res instanceof Object &&
7            'h' in res && 'm' in res);
8    ensures(res => Number.isInteger(res.h) &&
9            0 <= res.h && res.h < 24);
10   ensures(res => Number.isInteger(res.m) &&
11           0 <= res.m && res.m < 60);
12   return {
13     h: min / 60,
14     m: min % 60
15   };
16 }
```

Steps and Hints

(1) You need to add an additional precondition and change the returned value. (Hint: `Math.trunc` rounds a number down to an integer.)
(2) You can use the verification inspector and the editor counterexample popups.
(3) Click 'Next' if you fixed the example or if you want to finish the experiments.

B USER STUDY SURVEY ANSWERS

This appendix lists all survey answers by participants in the user study. Test subjects were given a series of online tutorials and programming tasks as listed in Appendix A, and then filled out an online survey. For features of the programming environment, participants could either select a given response or type their own answer.

Participant 1

JavaScript experience: 5/5 **Verification experience:** No

Verification Inspector	Did not use it	It impairs the development process
Counterexample Popups	Unsuccessfully tried using it	It could be helpful with different UI
Integrated Debugger	Unsuccessfully tried using it	The feature is helpful

Comments:

Participant 2

JavaScript experience: 5/5 **Verification experience:** Yes

Verification Inspector	Used this feature in experiments	The feature is helpful
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Used this feature in experiments	It could be helpful with different UI

Comments: I liked clicking on particular postconditions and being able to see counterexamples and add preconditions in the "scratch pad" area.

Participant 3

JavaScript experience: 4/5 **Verification experience:** Yes

Verification Inspector	Used this feature in experiments	The feature is helpful
Counterexample Popups	Did not use it	It could be helpful with different UI
Integrated Debugger	Did not use it	It could be helpful with different UI

Comments: JS is most useful on front end development. But how do you make assumptions/assertions for those UI/Networking related things?

Participant 4

JavaScript experience: 3/5 **Verification experience:** Yes

Verification Inspector	Used this feature in experiments	It could be helpful with different UI
Counterexample Popups	Did not use it	It could be helpful with different UI
Integrated Debugger	Did not use it	The feature is helpful

Comments: For a while I couldn't even tell that the debugging window was there. Perhaps have it always visible but only populated when something is selected.

Participant 5

JavaScript experience: 2/5 **Verification experience:** Yes

Verification Inspector	I tried to but was not successful.	It impairs the development process
Counterexample Popups	Unsuccessfully tried using it	It could be helpful with different UI
Integrated Debugger	Used this feature in experiments	The feature is helpful

Comments: I've never worked with assume or ensure before, and your modal text felt very jargon-y to me.

Participant 6

JavaScript experience: 5/5 **Verification experience:** Yes

Verification Inspector	I tried to but was not successful.	The feature is helpful
Counterexample Popups	Did not use it	It is not useful for development
Integrated Debugger	Did not use it	For more complex examples, but I can't judge the development experience as I did not use it.

Comments: I broke the interface when I tried to enter an assumption.

Participant 7

JavaScript experience: 5/5 **Verification experience:** No

Verification Inspector	I tried to but was not successful.	It could be helpful with different UI
Counterexample Popups	Unsuccessfully tried using it	It could be helpful with different UI
Integrated Debugger	Used this feature in experiments	It could be helpful with different UI

Comments: Keep up the good work!

Participant 8

JavaScript experience: 3/5 **Verification experience:** Yes

Verification Inspector	I tried to but was not successful.	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	It could be helpful with different UI
Integrated Debugger	Unsuccessfully tried using it	It could be helpful with different UI

Comments: The tutorial window often covered parts of the interface that I was required to interact with and there was no way to dismiss it which made it challenging to use the interface particularly in the example using the debugger. Entering assumptions in the right panel does not seem to add them or update the code in the editor which was confusing and makes the interface not seem very useful if they need to be entered in two separate places.

Participant 9

JavaScript experience: 5/5 **Verification experience:** Yes

Verification Inspector	Did not use it	The feature is helpful
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	The feature is helpful

Comments: counterexample of 1499 for min was super helpful!!

Participant 10

JavaScript experience: 1/5 **Verification experience:** Yes

Verification Inspector	I entered the ones from the tutorial as suggested	Yes, but it also calls for more logical foundation in introductory classes.
Counterexample Popups	I would have liked a cheat sheet for the parts that were not a tutorial. Maybe I'm a little tired and can't figure this out right now. Getting the answer like in the Tutorial would give me something to chew on.	The feature is helpful
Integrated Debugger	Used this feature in experiments	The feature is helpful

Comments: What I want to know as a lay person is will there be some AI auto-pilot who not only can tell me something is wrong, as this does, but pretty much gives me the right answer every time. That's what I want, a glorified spellchecker.

Participant 11

JavaScript experience: 2/5 **Verification experience:** No

Verification Inspector	Used this feature in experiments	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	It could be helpful with different UI

Comments: I didn't like the font in the UI. For instance the difference between == and === wasn't immediately obvious. For someone who doesn't program in JavaScript much this made the tutorial more difficult to follow. But overall it's an interesting prototype and interactive development of assertions along with hints from assertion editor seems like a useful tool for learning how assertions work and adding them in the code.

Participant 12

JavaScript experience: 5/5 **Verification experience:** Yes

Verification Inspector	Did not use it	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	It could be helpful with different UI

Comments: I had some UI difficulty with the specific implementation of adding assumptions (e.g. having to add them in multiple places) and another with the interface for the debugger being very narrow and requiring scrolling to see the whole thing. I did find this tool very compelling, although the hints to some extent obviated the need for the tool. It is very difficult to evaluate stuff like this.

Participant 13

JavaScript experience: 4/5 **Verification experience:** No

Verification Inspector	Did not use it	The feature is helpful
Counterexample Popups	Unsuccessfully tried using it	It could be helpful with different UI
Integrated Debugger	Did not use it	It could be helpful with different UI

Comments:

Participant 14

JavaScript experience: 4/5 **Verification experience:** Yes

Verification Inspector	Did not use it	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Unsuccessfully tried using it	The feature is helpful

Comments:

Participant 15

JavaScript experience: 4/5 **Verification experience:** No

Verification Inspector	I think I did on one of them but don't remember.	It is not useful for development
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	The feature is helpful

Comments: The counterexamples would be the most helpful contribution of this project to my own programming practice. It makes it easier and quicker to comprehensively test a function and catch edge cases, because I don't have to come up with the test values or edge cases myself. And it feels better to me to code pre and post

conditions explicitly, rather than write them in comments and reference comments or other documentation whenever I use the function in a sort of new way. I would want to write requires() and ensures() statements in my own Javascript programs.

Participant 16

JavaScript experience: 3/5 **Verification experience:** No

Verification Inspector	Unsuccessfully tried using it	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Used this feature in experiments	It could be helpful with different UI

Comments:

Participant 17

JavaScript experience: 4/5 **Verification experience:** No

Verification Inspector	They weren't visible clearly since the instructions/help panel was covering it	The feature is helpful
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	It impairs the development process

Comments: Great tool!

Participant 18

JavaScript experience: 4/5 **Verification experience:** No

Verification Inspector	Used this feature in experiments	It could be helpful with different UI
Counterexample Popups	Unsuccessfully tried using it	The feature is helpful
Integrated Debugger	Unsuccessfully tried using it	The feature is helpful

Comments: