Using Debuggers to Understand Failed Verification Attempts

Peter Müller and Joseph N. Ruskiewicz

ETH Zurich, Switzerland
{peter.mueller,joseph.ruskiewicz}@inf.ethz.ch

Abstract. Automatic program verification allows programmers to detect program errors at compile time. When an attempt to automatically verify a program fails the reason for the failure is often difficult to understand. Many program verifiers provide a counterexample of the failed attempt. These counterexamples are usually very complex and therefore not amenable to manual inspection. Moreover, the counterexample may be invalid, possibly misleading the programmer. We present a new approach to help the programmer understand failed verification attempts by generating an executable program that reproduces the failed verification attempt described by the counterexample. The generated program (1) can be executed within the program debugger to systematically explore the counterexample, (2) encodes the program semantics used by the verifier, which allows us to detect errors in specifications as well as in programs, and (3) contains runtime checks for all specifications, which allows us to detect spurious errors. Our approach is implemented within the Spec# programming system.

1 Introduction

A common approach to automatic program verification is to compute *verification conditions*, logical formulas whose validity entails the correctness of the program. The verification conditions are then passed to an automatic theorem prover, typically an SMT solver such as Simplify [7] or Z3 [6]. If the prover can establish the validity of the verification condition then verification succeeds; otherwise verification fails for one of the following reasons:

1. The program is incorrect, that is, the program does not satisfy its specification, and the *specification* expresses what the programmer intended. A typical example is a runtime error such as division by zero.
2. The specification is incorrect or incomplete, that is, the program does not satisfy its specification, and the *program* expresses what the programmer intended. A typical example is a loop invariant that is too weak.
3. The prover was too weak to validate the condition, that is, the verification error is a false positive, called a *spurious error*.

All three causes occur frequently in program verification; in particular, incorrect and incomplete specifications are as common as errors in programs. Spurious errors are less common, but are more difficult to understand when they do occur.

```
class IntList {
  int[] Elements;
  int Count;

  void Add (int value)
    modifies Count, Elements, Elements[*];
    ensures Contains (Elements, value)
  { ... }

  void Sort ()
    modifies Elements[*];
    ensures Sorted (Elements);
  { ... }
}
```

```
class SortedList {
  IntList list;

  // The list is sorted
  invariant Sorted (list.Elements);

  void AddSorted (int value)
    modifies list.Count, list.Elements[*];
    ensures Contains (list.Elements,value);
  {
    list.Add (value);
    list.Sort ();
  }
}
```

Fig. 1: Spec# is unable to verify `AddSorted`. The notation `Sorted(`a`)` abbreviates the condition that array a is sorted and `Contains(`a`,`v`)` abbreviates that v is contained in a. Both conditions can be expressed in Spec# via quantification over the indices of a. The modifies-clauses specify frame properties by listing the locations a method is allowed to modify. For brevity, we omit the method bodies in class `IntList`, access modifiers, as well as Spec#'s ownership and non-null annotations.

Consider the small Spec# [11] program in Fig. 1. The method `AddSorted` of class `SortedList` adds the `value` parameter to the list of integers and then sorts the list. The specification requires that after the execution of `AddSorted`, the list be sorted (by an object invariant) and that it contains `value` (by a postcondition). Verifying the method with the Spec# program verifier fails. A modular verifier such as Spec# verifies each method individually and reasons about method calls in terms of the callee's specification, not its implementation. The specification of `Sort` states only that the elements of the list will be sorted, not that they will be preserved. Consequently, the verifier is unable to prove that `value` is still contained in `list.Elements` after the call to `list.Sort` and, thus, that the postcondition of `AddSorted` holds. We will discuss a second verification error related to `AddSorted`'s modifies clause in Sec. 5.

A programmer who may not understand the cause of this failure from the program text can query the program verifier for a counterexample. The counterexample essentially contains a value for each variable in each state—a trace leading to the failing specification. For programs with non-trivial states (in particular, heap data structures) these counterexamples can be magnitudes larger than the program. For our example, the counterexample is over 1,200 lines of text. It is therefore not amenable to manual inspection and provides little benefit to the programmer. Moreover, due to the limitations of automatic proving, the counterexample may be invalid and not representative of a valid execution, thus misleading the programmer.

Using the initial state from the counterexample to construct a unit test for the failing method is helpful only if the error is in the program; errors caused by incorrect or incomplete specifications cannot be reproduced by a unit test. For example, a test that executes `AddSorted` with the initial state from the counterexample and then asserts the postcondition will succeed because the im-

plementation does satisfy its postcondition. It is the incomplete specification of `list.Sort` that causes the verification to fail, not the implementation. So successful tests are inconclusive about the presence and cause of verification errors.

In this paper, we present a technique that enables programmers to use standard debuggers to inspect program verification and counterexamples just as they use debuggers to inspect program executions and execution states. Our technique enables programmers to step through the verification of a method, check the validity of assertions, and observe the evolution of the state described by the counterexample. It detects verification failures caused by all three reasons mentioned in the introduction and notifies the programmer of invalid counterexamples. This tool support allows programmers to understand, locate, and fix verification errors more easily. We believe that applying a familiar tool for this task is crucial for making program verification more efficient and for increasing acceptance among practitioners. Our approach is implemented within the Spec# programming system. The tool, examples, and a demo video are available online at `http://www.pm.inf.ethz.ch/publications/cee`.

Outline. In Sec. 2 we give an overview of our approach. We explain how we reproduce counterexample states in Sec. 3. Sec. 4 describes how we rewrite the program to simulate its verification semantics and to reproduce the execution described by the counterexample. In Sec. 5 we extend the runtime assertion checker to handle all relevant specifications and show how it can be used to check the validity of the verification failure in Sec. 6. We discuss experiences using our approach and give a debugging procedure in Sec. 7. We present related work in Sec. 8 and conclude with Sec. 9.

2 Approach

Given a Spec# program and a counterexample produced by Z3, we construct an executable .NET program that simulates the verification semantics and reproduces states given by the counterexample. The constructed program can be executed in a program debugger, allowing the programmer to systematically and efficiently explore the counterexample. By executing the constructed program, we are able to detect spurious errors and validate failed verification attempts. The three key features of our approach are as follows:

(1) The constructed program *simulates the verification semantics* of the program as defined by the verifier rather than the concrete execution semantics as defined by the .NET platform. The semantics used by a program verifier is typically an abstraction of the execution semantics. Loops are typically verified via loop invariants rather than by considering the actual iterations, and modular verifiers reason about method calls in terms of method specifications rather than the implementation of the called method. By simulating the verification semantics rather than the execution semantics, we can detect verification errors caused by incorrect or incomplete specifications.

(2) The constructed program *reproduces the states given by the counterexample*. We execute the constructed program in the initial state described by the

counterexample. For each statement whose verification semantics differs from the execution semantics, we reproduce the effect of executing the statement by creating a program stub that alters the state as described by the counterexample. This allows programmers to use the debugger to explore and navigate through the counterexample.

(3) The constructed program *contains runtime checks for specifications* that are relevant for the verification error. For those specifications that generally cannot be checked efficiently at runtime (for instance, frame specifications, which universally quantify over all allocated objects), we use the counterexample to determine which objects are relevant for the verification error and focus the runtime checks on those. Moreover, checking the relevant specifications at runtime allows us to determine whether or not a verification error is spurious. This is the case if the constructed program terminates without a runtime error or specification violation.

Our approach enables the programmer to understand the failed verification attempt in method `AddSorted` as follows: We extract the initial state from the counterexample and construct a program driver that will create a `SortedList` object that contains an `IntList` object (in field `list`) with a list containing the elements, say, `0` and `1`. We then rewrite the body of `AddSorted` so that it simulates Spec#'s verification semantics. That is, we replace the calls to `Add` and `Sorted` with program stubs that change the program state to the state given by the counterexample. The stub for the call to `Add` changes `list.Elements` to contain the elements `[0,1,-3]`[1]. The stub for the call to `Sort` updates the state of `list.Elements` to some sorted array, say `[7,7,7]`. We finish by constructing a runtime check for the invariant of `SortedList` and the postcondition (and modifies clause) of `AddSorted`. For each step in the construction, we insert debugger directives that allow the programmer to control the execution of the of the *original* program, but observe the states of the *constructed* program.

A programmer using our approach is presented with the original implementation of `AddSorted` highlighted by the program debugger. The programmer can either use the debugger to inspect the initial (counterexample) state or execute the method until either the runtime assertion checker notifies them of a failing assertion or the method terminates, notifying the programmer of a spurious error. In our example, the runtime assertion checker will notify the programmer of the failing postcondition `AddSorted`, thus confirming the verification failure. The programmer can then inspect the post-state of the method and observe the value `[7,7,7]` for `list.Elements`. However, the initial state contained the state `[0,1]` for `list.Elements` and `-3` for `value`. The programmer can now single-step through the body of `AddSorted` inspecting the (counterexample) state of each step. Stepping over the call to `list.Add` adds `value` to `list.Elements`, as expected. Stepping over the call to `list.Sort` changes `list.Elements` to `[7,7,7]`. This unexpected change points the programmer to the cause of the verification failure, namely the incomplete specification of `Sort`. Note that it is

[1] Given the weak specification of `Add`, the counterexample could provide any array that contains the initial value of `value`, which we assume here to be `-3`.

the simulation of the verification semantics that enables us to identify the incomplete specification as the cause of this verification error. Using the execution semantics, for instance in a test case, could exhibit only errors in the code.

3 State Construction

To simulate the verification semantics of the failing method, we replace each statement whose verification semantics differs from the execution semantics by a program stub that alters the state as prescribed by the counterexample. Both for this purpose and to set up the initial state of the method execution, we extract information from the counterexample and construct the corresponding state.

A counterexample contains values for all local variables in each execution state; we use those to extract the method arguments. Moreover, it contains function interpretations, in particular, for the select and store functions that are used to encode the heap; we use those to extract field values. The extraction is relatively simple and works for all counterexamples

In this section we describe the construction of *mock types* that replace the original types in the program with versions that enable flexible initialization, of *program stubs* that construct the state given by the counterexample, and of the entry point to the failing method, the *driver*.

3.1 Type Mocking

For variables of built-in types such as primitive types and arrays, the state construction consists of straightforward assignments. For variables of user-defined types such as classes and interfaces, the state construction involves the creation of objects and the initialization of their fields according to the state given in the counterexample. Object creation is not possible for abstract types; initialization is difficult for types that do not provide a suitable constructor.

To address these issues we replace each user-defined type in the program by a mock type—a concrete class—that contains: (1) a parameterless constructor with empty body, which allows the program stubs to instantiate the class; (2) a declaration for each field that is accessible to the failing method or that is mentioned in a specification; if the field is of a user-defined type, we replace it by the corresponding mock type. We declare all fields of mock types public, which allows the program stubs to initialize them according to the counterexample via field assignments. Mock types do not contain any methods, except for the method that simulates the verification semantics of the failing method as we describe in Sec. 4.

In our example, we construct mock types for `SortedList` and `IntList`. Class `SortedList` contains a field `list`, which is accessed in the body of `AddSorted`. The type of this field is the mock type for `IntList`. All the fields of `IntList` are of built-in types. The type mocking is performed on the .NET level and transparent to the programmer.

3.2 Program Stubs

We replace each statement s whose verification semantics differs from the execution semantics by a program stub. This stub simulates the verification semantics of s by constructing the state after the execution of s as described by the counterexample. For this purpose, we extract the state before and after the execution of s from the counterexample. For each variable or field in which these two states differ, the program stub contains an assignment that updates the variable to reflect the state change.

When updating variables of reference types, we must preserve any alias properties contained in the counterexample, that is, when two variables contain the same symbolic reference in the counterexample, they must also contain the same reference in the constructed state. So when we update a variable of a reference type, we first check if we have already constructed an object for the symbolic reference in the counterexample. If so, we assign a reference to that object. If not, we create and initialize a new object, making use of the type mocking.

3.3 Driver

To begin executing the failing method we have to generate a *driver*, which constructs the initial state, attaches itself to the program debugger, and then calls the failing method. The initial state consists of values for the receiver, the method arguments, and all objects reachable from them (an extension to global data is straightforward). Its construction is a special case of the state construction described in the previous subsection; the only difference is that the driver constructs the entire state and not just the changes since a previous state. The programmer does not see the driver, but only the effects the driver produces.

```
// Construct the array for IntList.Elements
int[] Elements = new int[2];
Elements[0] = 0;
Elements[1] = 1;

// Construct an instance of IntList
IntList list = new IntList ();
list.Elements = Elements;
list.Count = 2;

// Construct receiver of failing method
SortedList rcvr = new SortedList ();
rcvr.list = list;
```

```
// Attach to the program debugger
Debugger.Launch ();

// Set the first step of the debugger
Debugger.Step ("rcvr.AddSorted (-3)");

// Call the failing method
rcvr.AddSorted (-3);
```

Fig. 2: The driver for our example first constructs the initial state, then launches the debugger, and finally calls the failing method. The types `IntList` and `SortedList` denote the mock types generated for the classes with the same names, which declare parameterless constructors and public fields.

The driver for our example creates the initial state for the failing method `AddSorted`, in particular, the receiver of type `SortedList` (Fig. 2, left column).

In order to initialize this object, it first constructs and initializes an `IntList` object that will be assigned to the receiver's `list` field. For this purpose, we create an integer array of the length given in the counterexample (`2`) and directly initialize its elements with the values from the counterexample (`[0,1]`). We use this array to initialize the new `IntList` object. After the initialization of the `IntList` object, the driver creates and initializes the receiver of the failing method.

After the initial state construction, the driver launches the debugger, and then calls the failing method `AddSorted` on the constructed receiver with the argument value from the counterexample, `-3` (Fig. 2, right column).

4 Verification Semantics

Program verifiers such as Spec# reason about a program using a verification semantics, which abstracts from the execution semantics. The two main abstractions are to reason about method calls in terms of the method's specification rather than its implementation (for the sake of modularity) and to reason about loops in terms of a loop invariant rather than actual iterations (to avoid impractical fixpoint computations). To help the programmer detect verification errors caused by incorrect or incomplete specifications, we replace in the failing method all method calls and loops by program stubs that simulate the verification semantics. The counterexample indicates which path through the failing method lead to the verification error; we use this information to eliminate all branches, jumps, and loops from the failing method. The resulting method body contains only straight-line code.

Although we rewrite the body of the failing method, the program debugger displays the original method body; the rewriting is transparent to the programmer. We achieve this effect by injecting debugger directives (in the form of calls to `Debugger.Step`) into the program stubs. These directives highlight the code in the original method body and allow the programmer to control the execution of the stubs from the original method body.

4.1 Method Calls

The verification semantics of a call to a method m is (1) to assert m's precondition, (2) to assign arbitrary values to all memory locations that may be changed by m (according to its modifies clause), and then (3) to assume m's postcondition. To simulate this semantics, we replace each call to a method m in the failing method, including recursive calls and constructor calls, with a program stub that contains: (1) a runtime check for m's precondition (2) code that updates the state of the program to reflect the state given by the counterexample as described in Sec. 3.2, and (3) a runtime check for m's postcondition; the motivation for this check will be explained in Sec. 6.

Method `AddSorted` contains calls to `list.Add` and `list.Sort`. For each call the counterexample contains a state describing the effect of the call. We replace

```
// Step over the method list.Add
Debugger.Step (list.Add);

// Construct the poststate of list.Add
int[] Elements = new int[3];
Elements[0] = 0;
Elements[1] = 1;
Elements[2] = -3;
list.Elements = Elements;
list.Count = 3;

// Check postcondition of list.Add
... // See Sec. 6.2
```

```
// Step over the method list.Sort
Debugger.Step (list.Sort);

// Construct the poststate of list.Sort
list.Elements[0] = 7;
list.Elements[1] = 7;
list.Elements[2] = 7;

// Check postcondition of list.Sort
... // See Sec. 6.2
```

Fig. 3: The program stubs replacing the calls to `list.Add` and `list.Sort` in the failing method `AddSorted`. The debugger directives instruct the program debugger to highlight the calls. The stubs construct the post-states of the calls given by the counterexample.

these method calls with the stubs in Fig. 3. The stub for the call to `list.Add` (left column) constructs the state as prescribed by the counterexample. In the counterexample, `Elements` contains a new symbolic reference; so we construct a new `Elements` array. The `list` field has not changed since the pre-state of the call, so we update only the state of the referenced object with the new values given by the counterexample. The stub then checks the postcondition of method `Add`, which we discuss in Sec. 6.

The stub for the call to `list.Sort` (right column) is analogous; however, we do not update `list.Elements` because the counterexample does not contain a value that is different from the pre-state (because the modifies clause of `Sort` does not permit modifications of the field `Elements`, only of the elements within the array). Note that the two stubs do not contain precondition checks because neither of the two methods has a precondition.

Specification languages such as Spec# allow specifications to contain calls to side-effect free (*pure*) methods. The verification semantics of such calls is to encode the pure method as a mathematical function that is axiomatized based on the specification of the pure method and not on its implementation [5]. Calls to pure methods in specifications are then encoded as applications of these mathematical functions. To simulate this semantics, we replace all occurrences of a pure method within a specification with the result value contained in the counterexample. Since pure methods are not allowed to change the heap, this simple replacement is sufficient to capture the effects of the pure method.

4.2 Loops

The verification semantics of a loop is: (1) to assert the loop invariant before the loop, (2) to simulate the state after an arbitrary number of (possibly zero) loop iterations by assigning arbitrary values to all locations that may be modified by the loop and assuming that the resulting state again satisfies the loop invariant. The verification semantics then considers two possibilities to continue the execution: (3) an arbitrary execution of the loop body by assuming that the condition of the loop holds, executing the loop body, and asserting that the loop

invariant holds again after the body, or (4) exiting the loop by assuming that the condition of the loop does not hold and proceeding to the statement after the loop. Checking an arbitrary iteration of the loop suffices to ensure that any execution of the loop preserves the loop invariant.

To simulate this semantics we replace each loop with a program stub that contains: (1) a runtime check for the loop invariant, and (2) code that updates the state of the program to reflect the state given by the counterexample as described in Sec. 3.2 and another runtime check for the loop invariant, which we discuss in Sec. 6. From the counterexample, we know whether the verification error occurred on the path that contains the arbitrary loop iteration (branch (3)) or the path that exits the loop (branch (4)). In case (3), the stub contains a runtime check for the loop condition (see Sec. 6), the loop body (replacing any method calls or inner loops), another runtime check for the loop invariant, and then terminates the execution of the method. In case (4), the stub just contains a runtime check for the negation of the loop condition (see Sec. 6) and then proceeds with the code following the loop.

As we mentioned above, a programmer using our approach will not see the program stubs, but only the effect they have on the state of the program. If the error is located in the loop body, the execution as presented to the programmer enters the loop body; upon entry, the programmer will observe a sudden change of the state to the arbitrary state prescribed by the counterexample (satisfying the loop invariant and the loop condition). If the error is located after the loop, execution skips the loop entirely, also with a sudden change of the state (to an arbitrary state that satisfies the loop invariant and the negation of the loop condition).

5 Extended Runtime Checking

We rely on the runtime assertion checker to reproduce failed verification attempts. An execution of the rewritten failing method that does not lead to an assertion violation indicates a spurious error. To be conclusive about a verification failure, the runtime checker must be able to check any failing assertion.

Most assertions in Spec# programs are executable. In particular, quantifiers that range over finite integer intervals, such as array indices, are checked by iterating over the range. However, the verification semantics of Spec# also makes use of assertions that quantify over possibly unbounded sets, for instance, over all allocated objects in the assertions for modifies clauses and object invariants. Such assertions cannot be checked efficiently at runtime.

Nevertheless, we can generate useful runtime checks for most *failed* quantified assertions. When an assertion with a *universal* quantifier fails to verify, the counterexample contains instantiations of the quantified variables for which the assertion does not hold. In order to check whether a verification error is spurious, it is sufficient to generate a runtime check for those specific instantiations, which is straightforward. For unbounded *existential* quantifiers, the counterexample does not contain useful information because one would have to check all values

of the unbounded set, not just one. However, automatic program verifiers avoid unbounded existential quantifiers because they are not handled well by SMT solvers. Therefore, not checking them at runtime is not a limitation in practice.

In our example, method `AddSorted` does not satisfy its modifies clause because the call to `list.Add` may modify `list.Elements` but `AddSorted` must not. Therefore, the static verification of `AddSorted` leads to a second verification error. The counterexample for this error contains instantiations for the quantified variables in the assertion for `AddSorted`'s modifies clause. Here, these instantiations indicate that the `Elements` field of the object `list` is being modified without permission by the modifies clause. Using this information, we generate code that stores the initial value of `list.Elements` upon entry to `AddSorted` and then checks that `list.Elements` has not been modified upon termination of the method. Since the program stub for the call to `list.Add` changes the value of `list.Elements` (see Fig. 3), this runtime check fails and confirms the verification error.

The programmer debugging this verification failure can localize the error efficiently by attaching a data breakpoint to `list.Elements`. If a statement then modifies `list.Elements`, the debugger stops the execution notifying the programmer of the modification; in our example, at the call to `list.Add`. The programmer, now aware of the location of the failure, can fix the error by weakening the modifies clause of `AddSorted`.

6 Error Validation

In this section we explain how our approach detects spurious errors and invalid counterexamples.

6.1 Spurious Errors

Since the validity of verification conditions is undecidable, SMT solvers cannot always determine whether a verification condition is valid or not. Whenever the SMT solver does not provide a conclusive result, a sound verifier needs to be conservative and report a verification error, which is possibly spurious. Spurious errors occur frequently in automatic program verification, for instance, when specifications include quantifiers or non-linear arithmetic.

By extending the runtime assertion checker to handle *all relevant* failing assertions in Spec#, we are able to validate verification failures. If the execution of the rewritten failing method terminates without a failed runtime assertion check, we can safely conclude that the error is spurious and notify the programmer; who can now address the problem by rephrasing the specification, rather than spending time determining the cause of an error that does not exist.

6.2 Invalid Counterexamples

A counterexample is supposed to satisfy all assumptions that are being made in the verification semantics of a program. For instance, the initial state in a

counterexample is supposed to satisfy the precondition of the failing method. However, if the assumptions contain formulas that are beyond the capabilities of the prover, it might construct an invalid counterexample that contradicts the assumptions. For example, most automatic provers do not fully support non-linear arithmetic and might produce an initial state such as `-563` for `x` and `4` for `y` for the precondition `x / y > 0`. Simulating the execution described by an invalid counterexample and, in particular, checking assertions in states extracted from an invalid counterexample, is not helpful to understand verification errors.

We extract states from the counterexample in three cases: (1) to set up the initial state in the driver, (2) to reproduce the state changes made by a method call, (3) and to reproduce the state changes made by a loop iteration. For these cases, the verification semantics of Spec# makes the following assumptions about the expected state: (1) the precondition of the failing method, (2) the postcondition and modifies clause of a called method, and (3) the loop invariant and the loop condition. To guard against invalid counterexamples, we introduce runtime checks for each of these assumptions. When such a runtime check fails, it indicates that the counterexample state does not satisfy the assumption and, thus, the counterexample is invalid.

The failing method `AddSorted` of our example assumes its precondition as well as the postconditions and modifies clauses of the called methods `Add` and `Sort`. The assumption for the precondition would be part of the driver, which extracts the initial state from the counterexample, but is omitted in Fig. 2 because `AddSorted` has no precondition. The assumptions for the postconditions are part of the program stubs that replace the calls. To the stubs in Fig. 3, we append the checks `assert Contains(list.Elements,value)` and `assert Sorted(list.Elements)`, respectively.

Our approach checks most assumptions in the verification semantics at runtime, but not all of them. Assumptions that are not checked include for instance the modifies clause of a called method, which contains an unbounded universal quantification; the extended runtime checking described in Sec. 5 does not apply here, because this check does not correspond to a failed assertion and, therefore, the counterexample does not provide instantiations for the quantifier. Therefore, our approach might theoretically miss some invalid counterexamples, but that has not happened in any of the examples we have tried so far.

7 Experience

We have applied our approach in debugging the various verification failures found in examples from the Spec# tutorial [11], the Spec# test suite (see `http://specsharp.codeplex.com`), and our own test suite[2]. In this section, we outline a systematic procedure that we have found to be effective for using our approach to locate the cause of verification failures. We also summarize and evaluate our experiences using this procedure.

[2] Also included in the download of our tool.

The main observations of our experiments are: (1) Our approach is helpful for understanding most of the verification failures in the examples. In particular, we were able to effectively and efficiently detect bugs in the implementation as well as incorrect or incomplete specifications. The examples where our approach did not provide any benefit were fairly obvious errors in small methods. For those verification failures, the error message provided by Spec# was sufficient to localize and fix the error. (2) Our set of examples contained very few spurious errors and invalid counterexamples because we took them mostly from the Spec# tutorial and test suite, both of which focus on examples that are handled well by the verifier. Nevertheless, our runtime checks identified all of the spurious errors and invalid counterexamples. (3) Most verification failures can be debugged systematically with a simple procedure, which we outline below.

These initial results are very promising. However, our evaluation may be biased in two ways. Firstly, the examples were written for Spec# demonstrations and might not be representative of real application code. Secondly, the evaluation was performed by people who are familiar with Spec#'s program verifier; it is possible that programmers might struggle with issues that are obvious to us. Nevertheless, we are confident that our positive experience will be confirmed by programmers working on application code.

Debugging Procedure. We have found the following steps to be an efficient way to localize and understand the cause of a verification failure. If the verifier reports several errors for the same method, we debug them in the order of their source location.

1. Use the error message to check the method for obvious errors. For very simple programs and specifications our approach usually requires more effort than simply inspecting the failing method. This is often the case for programs that contain neither method calls nor loops, which reduces the likelihood that the verification failure is caused by an incorrect or incomplete specification.

2. Run the rewritten program in the debugger and observe the failure. Before attempting to localize the error, one should first confirm that the verifier has found a valid error by running the rewritten program in the debugger. This run will either result in an assertion violation (confirming the validity of the error), in a failed assumption check (indicating an invalid counterexample), or in a message that suggests that the error is spurious. In the latter two cases, the programmer needs to find an alternative way of expressing the program or its specification and re-verify the program. In the former case, the debugging procedure continues with the next step.

3. Inspect the state in which the assertion failed. The runtime check for an assertion fails either because the assertion is incorrect or because the assertion was checked in a state the programmer did not expect. We recommend to inspect the assertion and the state in which the runtime check failed to determine which case applies. If the assertion is incorrect, we can fix it and re-verify the method. If the state contains unexpected values, we determine their origin in the next step.

4. Step through the rewritten program and observe changes to the relevant variables. From step 3, we know which assertion fails. It is helpful to track the values of the variables in this assertions to detect unexpected values, for instance, caused by a weak precondition or loop invariant. This tracking is best performed by adding these variables to the variable watch window of the debugger and then single-stepping through the rewritten method. Unexpected initial values point us to a weak precondition; unexpected modifications during a single step require further investigation, described in step 5. Single-stepping through the method is also likely to reveal errors in the code such as incorrect control flow or the absence of a necessary assignment.

A variation of step 4 is more efficient when the failing assertion contains only a small number of variables, such as the runtime check for a modifies clause which focuses on only one heap location (see Sec. 5). In this case, one can avoid the single-stepping and instead add data breakpoints for the relevant variables. We can then run the rewritten method in the debugger and get notified whenever a variable of interest gets updated.

5. Analyze unexpected modifications. Step 4 determines where a variable receives an unexpected value. If this happens during a method call or in a loop, we have identified the method's specification or the loop invariant as the cause of the unexpected value and can amend them. If the unexpected value comes from an assignment then we may also need to track the variables in the right-hand side expression by adding them to the watch window and repeating from step 4.

8 Related Work

The literature contains several proposals for extracting useful information from counterexamples, but in the context of deductive program verification, these proposals are generally not sufficient to understand the verification failure. In particular, they do not support the programmer in detecting incomplete specifications, spurious errors, and invalid counterexamples.

Some verifiers such as Spec# apply heuristics to extract those parts of a counterexample that are likely to be relevant for the verification error. However, it is difficult to tune the heuristics such that they provide all necessary information without swamping the programmer with irrelevant details. For instance, Spec# filters too aggressively for the method `AddSorted` and it provides only the following excerpt from the counterexample, which does not point us in the direction of the error: `(initial value of: value) == -3`.

Trace and distance based techniques [1,8,3] have been applied successfully in the context of model checking to localize program errors. They compare successful program executions against failing executions to determine which branches of the program lead to the error. Narrowing down the location of the error is useful, but may not suffice to determine the actual cause of the error. For instance, since the method body of `AddSorted` does not contain branches, these techniques will not provide any benefit. They also do not assist the programmer in detecting spurious errors. Another localization technique is program slicing

[14], which systematically removes statements that are not relevant for the validity of the failed verification condition. In practice, however, program slicers do not effectively reduce the size of programs (and counterexamples) with heap data structures and specifications containing quantifiers. Slicing the body of `AddSorted` will not result in a smaller program because both statements affect the state of `list`, which is relevant for the failing postcondition.

Another approach is to construct a test case from a failed verification attempt, using the initial state of the counterexample as test input [4,2,13]. This approach is only helpful if the test leads to a runtime error or if the violated specification can be found by a runtime assertion checker. However, when static verification fails because of incomplete specifications, or when the violated specification is not checked at runtime (for instance, when the specification contains unbounded quantification over objects), or when the error is spurious, the test case will succeed and, thus, not help the programmer to determine the cause of the verification error and might even mislead the programmer into believing that the error does not exist [4].

Verification techniques based on symbolic execution assist the programmer in understanding failed verification attempts by presenting the programmer with the symbolic states used during the verification process [9,10]. Inspecting a symbolic state is very helpful to a verification expert who is familiar with the symbolic representation of the program, whereas our approach seems more appropriate for programmers. Moreover, it is not clear to what extent symbolic states help in detecting spurious errors.

Alternative techniques based on visualizing the counterexample, such as those based on graph visualization [12,15], are limited by the size of the state presented and do not help in identifying spurious errors and invalid counterexamples.

9 Conclusions

We have presented our approach to help programmers to understand failed verification attempts. We generate an executable program that reproduces the verification error by encoding the verification semantics of the program and by using variable values from a counterexample. We extend the runtime assertion checker to reproduce all relevant verification errors, identify spurious errors, and detect invalid counterexamples. Executing the generated program inside a debugger allows the programmer to systematically and efficiently explore the counterexample; which is crucial for understanding, localizing, and fixing the verification failure. The generation of the executable program is entirely automatic and is transparent to the programmer.

We have implemented our approach in Spec#, but it is applicable to all program verifiers based on automatic provers that provide counterexamples. Our experience using our approach is very promising; we are able to understand and fix verification errors effectively and efficiently. As an additional benefit, we have found our approach useful to debug the encoding of Spec#. We have indeed found an error in the Spec# verifier; when inspecting a counterexample in our tool, we

noticed that a variable of type `uint` contained a negative value, which pointed us to an omission in the encoding of Spec# programs.

The main direction for future work is to combine our approach with counterexample-based dynamic program slicing to further reduce the time for localizing and fixing verification errors. Slicing will in particular allow us to automate step 5 of our debugging procedure.

Acknowledgments. We are grateful to the reviewers for their insightful comments. We would like to thank Christoph M. Wintersteiger for the various discussions on the internals of SMT solvers. We are also indebted to Jürg Billeter for the initial implementation of the tool and Christoph Studer for adding additional support for pure methods and modifies clauses.

References

1. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, pages 97–105. ACM, 2003.
2. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumbar. Generating tests from counterexamples. In *ICSE*, pages 326–335. IEEE, 2004.
3. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
4. C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.
5. Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
7. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories, Palo Alto, 2003.
8. A. Groce. Error explanation with distance metrics. In *TACAS*, volume 2988 of *LNCS*, pages 108–122. Springer, 2004.
9. R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A visual interactive debugger based on symbolic execution. In *ASE*, pages 143–146. ACM, 2010.
10. R. J. Hall and A. Zisman. Validating personal requirements by assisted symbolic behavior browsing. In *ASE*, pages 56–66. IEEE, 2004.
11. K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In *Advanced Lectures on Software Engineering*, volume 6029 of *LNCS*, pages 91–139. Springer, 2010.
12. D. Rayside, F. S.-H. Chang, G. Dennis, R. Seater, and D. Jackson. Automatic visualization of relational logic models. *ECEASST*, 7, 2007.
13. N. Tillman and W. Schulte. Mock-object generation with behavior. In *ASE*, pages 365–368. IEEE, 2006.
14. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.
15. A. Zeller and D. Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, 1996.