Counterexample Generation for Formal Verification of ABS

Bachelor thesis by Nils Rollshausen
Date of submission: 30.10.2020

1. Review: Eduard Kamburjan
2. Review: Richard Bubel
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Software Engineering Group

Counterexample Generation for Formal Verification of ABS

Bachelor thesis by Nils Rollshausen

1. Review: Eduard Kamburjan
2. Review: Richard Bubel

Date of submission: 30.10.2020

Darmstadt

Contents

1 Introduction

While formal verification is of interest for many applications and programming languages, it is perhaps especially useful for a language like the *Abstract Behaviour Specification Language* (ABS) [16] specifically designed for specification and modelling of complex systems. It is only natural, for example, to prove safety and security properties for protocols, be they cryptographic or operational, modelled in ABS. To this end, the *crowbar* tool [18] can execute ABS programs symbolically and prove correctness with regards to specification using common automated theorem provers like *z3* [20] or *cvc4* [4].

However, formal verification of a program is rarely a one-off process. In many cases, the initial specification or implementation is faulty, making too strong assumptions or too weak guarantees. This makes specification and verification an iterative process requiring frequent human intervention [5]. Here, the use of automated theorem provers, which enabled the verification of complex programs in the first place, becomes an issue: The generated proofs are usually much too large and complicated to be easily understood by a human. This makes 'debugging' uncloseable proofs both time-consuming and frustrating.

Despite the iterative nature of the verification process, most formal verification tools are designed for a more linear workflow, failing to support users in the debugging process. The crowbar tool, for example, returns only a simple boolean value to indicate the success of a proof by default. To understand *why* a proof failed, one has to examine the crowbar-internal proof tree and the output from the external theorem prover, searching for some discrepancy between the program logic and the specified behaviour.

Drawing a parallel to the general field of programming — also iterative by nature — shows that the tools available to developers do not have to be so limited to a linear perspective: While early text editors and compilers might have been similar to the tools available in formal verification today, we now have access to advanced debuggers, IDEs performing static analysis in real time as we type, and compilers giving extensive error messages and warnings. So even though it is rare to successfully compile a complex program on the first attempt after significant changes, there is sufficient tool support available to iterate on these changes and come closer to a working solution.
Why, then, are we still stuck with boolean return values on failed proofs in crowbar?

In this work, we will explore one possible approach to improving the proof failure case in crowbar by providing counterexamples showcasing specification-breaking behaviour.

1.1 Problem Definition

With the motivation for our method laid out, we can now define the problem as follows:

Given a method[1] M that failed verification, we want to derive another method M', called the counterexample, that illustrates an execution path of M that leads to specification-breaking behaviour. More specifically, the generated counterexample should satisfy the following informal requirements:

1. Minimal size. The generated counterexample should be as small as possible and reproduce only relevant parts of the original method. This makes the counterexamples easier to read and understand for human users.
2. Minimal dependencies. The generated counterexample should require as little external context — such as method or class definitions — as possible. This reduces the prior knowledge necessary to analyze counterexamples and increases their portability.
3. High abstraction. Counterexamples should hide as much of the internal proof complexity as possible. This makes the counterexamples accessible to users who are less familiar with the inner workings of the tools they use.
4. High clarity. Counterexamples should be easily understood and analyzed by human programmers.

Guided by this problem definition, we will continue to outline a high-level approach to counterexample generation and describe its implementation and integration into the *crowbar* tool as a module that we call the *crowbar investigator*. The implementation of the crowbar investigator is available online at `https://github.com/rec0de/crowbar-tool`.

1.2 Related Work

To the best of our knowledge, there is no existing work on counterexample generation in the context of formal verification that quite matches the scope of our proposed method.

The closest in spirit are methods that aim to provide a better debugging experience for formal verification, in most cases by providing tools inspired by interactive debuggers for programming. While there are several such tools for different types of verification tasks [12, 13, 5], the one most relevant for our work is the *Interactive Verification Debugger* (IVD) by Hentschel et al. [12]. Designed as a plug-in for the *KeY System* [2], it aides the programmer in performing deductive verification of Java programs by providing an interactive view of the proof tree in which proof issues are highlighted and can be corrected. The high-level goal of the IVD is thus very similar to the one of our proposed method. A key difference, however, is that whereas the IVD only provides a different view of the proof tree, which is already available to the user in other places throughout the KeY tool, the crowbar investigator generates an entirely new representation of issues within the proof. The main advantage here is that the generated counterexamples require less knowledge of the internal proof structure and logic formalization.
The KeY system also already provides a counterexample generation option — despite the similarity in name to this work, however, the counterexamples generated by KeY are rather rudimentary, providing only a shallow abstraction over the model provided by an SMT solver.

While the above methods share the motivation behind the crowbar investigator, the techniques they apply are very different. There are, however, related methods to be found in other fields, such as *testcase generation* and *exploit generation*.

[1] While we focus on methods for this work, other language constructs such as *init* blocks are treated like methods by crowbar and are thus handled in the same way.

Testcase generation methods commonly use bounded symbolic execution with loop unrolling to obtain a set of feasible execution paths through the input program [8]. They then use the branch conditions along every branch in the tree to generate input vectors that will trigger these execution paths, thus creating test data with a high code-coverage. Some testcase generation methods can then check the output produced by the tested code for each input vector against a formal specification of the desired behaviour [14, 8]. These methods are mostly related in how they use symbolic execution to generate data for testing and debugging purposes — however this data is usually limited to input/output vectors for the program under investigation, treating it essentially as a black box. Due to the different requirements for our method, we will go beyond such test vectors and generate modified versions of the original program.

Most similar to our method is a work on exploit generation by Do et al. [7]. Just like the testcase generation methods we have seen above, the method uses symbolic execution to generate a symbolic execution tree, which is then searched for behaviour that breaks previously specified information flow security guarantees. Unlike most testcase generation methods, however, Do et al. use formal specification for loop invariants rather than bounded unrolling.
What differentiates our method from this approach is that, like most testcase and exploit generation methods, its output is just a vector of input data for the examined method which may produce unwanted behaviour. With the crowbar investigator on the other hand, we are actually extensively modifying the method source code to account for proof failure-cases that do not directly translate to input data producing faulty behaviour.

2 Preliminaries

As we are building the crowbar investigator on top of several existing technologies, we will take some time here to introduce relevant concepts:

2.1 ABS

The *Abstract Behaviour Specification Language* or ABS [16] is a modelling language designed to formalize concurrent and distributed systems. While our method does not require in-depth knowledge of ABS, the high-level design concepts of the language and its concurrency model will influence the counterexample generation in several ways. The following summary is based on the original ABS paper [16].

At its core, the concurrency model of ABS is based on *concurrent object groups* or *cogs*. Each cog is a group of one or more objects with a dedicated processing unit and local memory. Within a cog, only a single process (e.g. a method invocation) can be executed at any given time and control of the processing unit is managed in a cooperative and non-preemptive manner — an active process will execute until it either terminates or voluntarily releases control, for example to wait for results from other processes to become available. This means that all concurrency in ABS has to be modelled explicitly.

To model true parallelism, then, we have to consider objects distributed across several cogs, with one active process in each cog. Communication and synchronisation between cogs is only possible using asynchronous method calls between objects. These calls take away control from neither the caller nor the callee, but simply create a new, dormant process in the callee's cog that will handle the method invocation. This process can be scheduled whenever the currently active process releases control, although all scheduling is non-deterministic and any dormant process may be chosen for execution. The caller receives a *future* as a placeholder for the pending result of the method call and can immediately continue execution. When the caller needs to access the call's return value, they can release control of the processing unit and wait for the future to resolve when the process on the remote cog has finished execution.

As each cog has its own local memory, processes on other cogs cannot access or modify any of its memory locations directly and concurrent processes cannot interfere with each other.

To illustrate the unique aspects of the ABS concurrency model, we will take a moment to discuss the example in Listing 2.1. This example shows a simple *reader* class which can query different temperature sensors and keep track of the highest measured temperature. The two sensors are each created in their own cog and could be used by several clients, although this is not shown in the example.
Let's assume a reader object receives instructions — from unspecified third parties — to read temperatures from both sensors simultaneously. Both of these requests would result in a new process being created in the reader's cog to service the method call to `read`. One of these processes will be scheduled first and initiates an asynchronous call to its associated sensor in line 11. The sensor in question, however, might currently

be busy serving other requests. Instead of a return value, the reader obtains a future that references this specific asynchronous call and can be used to obtain its result once it becomes available. The reader process could now continue executing instructions that do not depend on the temperature reading. In this case, the reader chooses to wait for the value to become available (line 12). While the first reader process waits, it releases control of the processing unit, meaning that the second process can be scheduled and in turn issue its asynchronous call to the sensor. In the meantime, the first sensor might have completed the request to get a new temperature reading, meaning that the future obtained by the first process resolves and the process becomes eligible for execution again. When the second reader process then waits for its future to resolve in line 12, the first process can be scheduled again, receive the temperature from the sensor reading (line 13), and update the highest recorded temperature accordingly (line 14–16).

```
1  interface Sensor {
2    Int getReading();
3  }
4
5  class Reader {
6    Sensor sensor1 = new TempSensor();
7    Sensor sensor2 = new TempSensor();
8    Int maxTemp = 0;
9
10   Unit read(Sensor s) {
11     Fut<Int> reading = s!getReading();
12     await reading?;
13     Int temp = reading.get;
14     if(temp > this.maxTemp) {
15       this.maxTemp = temp;
16     }
17   }
18 }
```

Listing 2.1: A simple sensor / reader example for concurrent ABS.

Note how the explicit modelling of concurrency and interleaving avoids race-conditions here: In common concurrent programming languages, lines 14 to 16 would introduce a race-condition when an interleaving of two reader-processes occurs after line 14. In this case, the first process has determined that it has recorded a new temperature-high and will write it to the `maxTemp` field as soon as it is re-scheduled. If the second process has recorded an even higher temperature and writes it to the field before the first process is re-scheduled, this update will be lost. However, as ABS uses explicit, cooperative scheduling and only one process can be active in a given cog at a time, these issues cannot occur in our example. The only times at which `maxTemp` can be modified by other processes in the same object are points at which a process voluntarily gives up execution such as in line 12.

The ABS concurrency model therefore provides a high degree of isolation between processes, which significantly simplifies formal verification and counterexample generation as outside interference can only occur at specific points in the program.

2.2 Symbolic Execution

Symbolic execution is a common method used in formal verification of software. It refers to simulating the execution of a program using symbolic — i.e. unknown — input values. The crowbar tool in particular uses KeY-style symbolic execution as described in the KeY book [2, pp. 385–390]. In contrast to regular execution, this produces not a single execution path but a tree of execution paths: Whenever the executed program makes a branch decision based on a value that is only symbolically known, the symbolic execution has to consider both possible paths, thus splitting the execution tree. During the symbolic execution, any state changes are tracked in some formalization of the program memory. When the program is completely executed, this formalization, together with other premises collected during execution, can be used to reason about properties of the input program (i.e. showing that a certain memory location can never be zero).

A symbolic execution tree for a method invocation of `read` from our previous example in Listing 2.1 might look something like the tree in Figure 2.2. Note that this example and the notation within it is kept very informal and is only intended to illustrate the basic concept of symbolic execution.

In this example, the execution tree splits once in node 4 when executing the if-statement.[1] Since we do not know concrete values for `temp` and `maxTemp`, we cannot tell which branch of the statement will be executed. Therefore, we split the tree into two branches — in one branch, we assume that the branch condition is true, in the other we assume the opposite. In this case, one branch assumes that we have measured a new temperature-high while the other assumes the measured temperature was below the previously recorded maximum.

Once all branches are completely executed, we can use the information collected in the leaf nodes — shown in grey in the example tree — to argue about the program. Since every possible execution path leads to one of these leaves, showing that the information in each leaf implies a certain property is sufficient to show that the given property holds for all executions.
In this case, we might want to show that calling `read` never decreases the recorded maximum temperature. If we look at the first leaf in the execution tree, we can see that its premises include temp $>$ maxTemp and heap.maxTemp $:=$ temp, meaning that the value of `maxTemp` is set to `temp`, which we know is larger than the old value of `maxTemp`. In the other leaf, the value of `maxTemp` is never modified, which appears to satisfy our claim, proving that `maxTemp` is never decreased.

This apparent proof, however, is incorrect: It is easy to overlook that the *await* statement releases control of the processing unit, thus allowing modifications to the heap while the process is dormant. In fact, the information collected in node 3 in the figure includes heap $:=$ anon(heap), indicating this loss of information about the heap. This means that while we can show that the method never decreases `maxTemp` below the value it had in the heap after the *await* statement, another process may decrease `maxTemp` while our process waits for the sensor reading to become available.

2.3 SMT Solvers

To make arguments about the program based on formulas obtained from symbolic execution, we need some form of an automated theorem prover. In this work, we will use the family of *SMT solvers* specialized on the

[1] Realistically, the tree would split once more when executing the asynchronous call in node 1 since the variable `s` might be `null`, causing an exception.

problem of *satisfiability modulo theories*. These solvers typically use the standardized input and output format *SMT-LIB v2* [3]. This gives us the flexibility to use any of the compatible solvers with no further modifications to our method. Two popular solvers using the SMT-LIB format are *z3* [20] and *cvc4* [4].
Given an input formula, the SMT solver will return a satisfiability result of either *sat*, *unsat*, or *unknown* in case the execution times out before completing. Also defined in the SMT-LIB standard is the *get-model* command which we will exploit to generate counterexamples. For a satisfiable input, the get-model command returns a set of interpretations for free variables and functions that satisfies the formula.

2.4 crowbar

The *crowbar* tool [18] is the symbolic execution engine and formal verification tool for ABS that we will be extending with counterexample generation capabilities. Crowbar extracts specification from annotated ABS code and symbolically executes the program using a small set of pre-defined rules. More specifically, crowbar uses the *behavioural program logic* (BPL) as described in [17] and a calculus similar to the one presented therein to execute ABS code and verify that its behaviour matches a given specification.
A proof of a program specification in BPL is a proof of the *behavioural modality* $[s \Vdash \tau]$, intuitively stating that every possible execution of the program s satisfies the specification τ. While this specification can express complex properties over possible execution runs of the program [17, p. 392], it is sufficient for our purposes to think of τ as a set of postconditions to the program execution.

Similar to the *dynamic logic* used by the KeY System [2, pp. 49–60], BPL uses *updates* to keep track of state changes during program execution [17, p. 398]. Looking at our running example from Listing 2.1 again, we can see how updates can be used to track state changes when symbolically executing the assignment in line 15. Let's assume that we have already executed the prior lines and arrived at the symbolic state $\{U\}[\text{this.maxTemp} = \text{temp}; s' \Vdash \tau]$, where U is some update and s' is the remaining program. We can now apply the rule for assignments to fields, obtaining the new state $\{U\}\{\text{heap} := \text{store}(\text{heap}, \text{maxTemp}, \text{temp})\}[s' \Vdash \tau]$, where the new update indicates that the field `maxTemp` on the heap is assigned the value `temp`. If `maxTemp` were a local variable, we would simply write $\{U\}\{\text{maxTemp} := \text{temp}\}[s' \Vdash \tau]$ instead. A more detailed explanation of updates and rules for their simplification can be found in [2, pp. 57–60].

The symbolic execution process then starts with the modality we are attempting to show: $[s \Vdash \tau]$, where s is the entire method under verification. Crowbar then successively applies rules from its calculus that execute the outermost statement of s, iteratively modifying s and — potentially — τ and accumulating state changes in updates.

For each leaf in the resulting symbolic execution tree, crowbar computes pre- and postconditions in standard first-order logic, modelling the requirements of the specification. These are then fed into a dedicated SMT solver to show that the precondition implies the postcondition. Specifically, crowbar attempts to show that leaf formulas are universally true by proving that their negation is unsatisfiable. Each leaf for which this proof succeeds is considered closed. If all leaves in the execution tree can be closed, the verification succeeds and the input program is proven correct with regards to the specification. A leaf that cannot be closed represents some execution path through the program that violates the specification. These are the paths that we will attempt to find and reconstruct for our counterexample generation method.

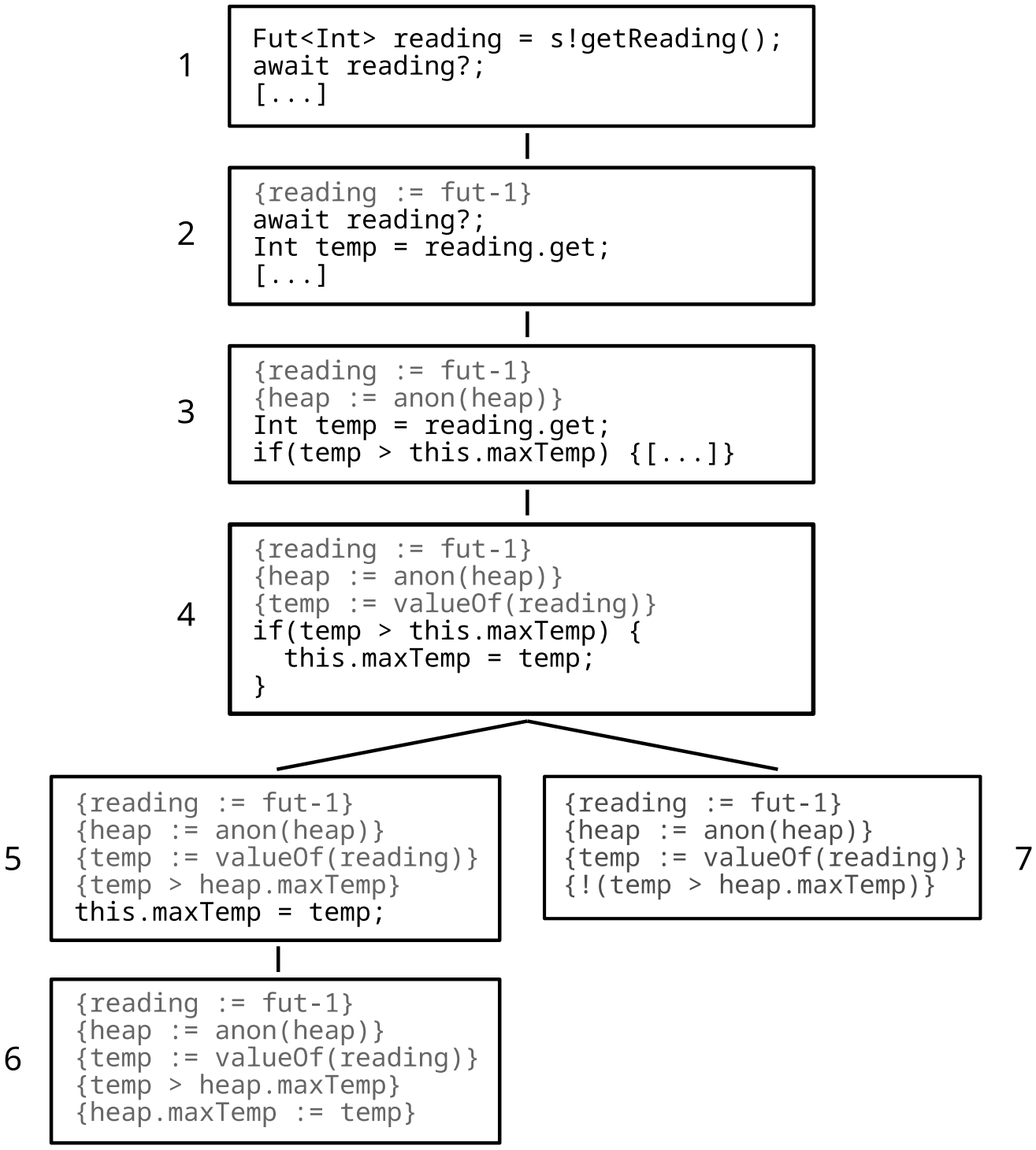

Figure 2.2: An informal, schematic symbolic execution tree for method `read` from Listing 2.1. Statements yet to be executed in black, gathered information in grey.

3 Method

In the following, we outline the abstract method used by the crowbar investigator to generate counterexamples for ABS. This method is independent of the specific implementation and can be adapted to most languages and symbolic execution engines. Specifically, our method requires that a symbolic execution engine is used in conjunction with an off-the-shelf SMT solver. Adaption to other systems that mix symbolic execution and theorem proving, while possible, would likely require significant modifications.

Fundamentally, any counterexample generation method in the context of a symbolic execution engine and a separate theorem prover can draw information from three sources: The original program source code, the symbolic execution tree, and the counterexample or model provided by the theorem prover.

However, as the complexity of aligning the source program — for example in the form of an *abstract syntax tree* (AST) — to the symbolic execution tree and the logic model is quite large, we will not use source code information explicitly in the proposed method. Instead, we make use of the fact that the symbolic execution engine already traverses the relevant parts of the source AST during symbolic execution.

By modifying the symbolic execution engine to store some additional information in the symbolic execution tree, we can use the resulting tree to reconstruct the executed program. Using this approach, the stored source code information is inherently linked to the relevant symbolic execution node, allowing us to easily combine information from specification and source code during the counterexample generation.

It is worth noting that this is not necessarily an optimal approach: The tight coupling of the symbolic execution engine and the counterexample generator makes the method less general and more sensitive to changes in the execution engine. While it is possible to imagine a more universal counterexample generator that takes a raw symbolic execution tree as an input, it is hard to justify the associated increase in complexity in our scenario.

3.1 Program Reconstruction

Any failed verification attempt will result in a symbolic execution tree with one or more unclosed leaves. Each of these leaves represents a different execution path of the program for which correctness could not be shown. As each of these paths may be uncloseable for different reasons, we will generate counterexamples for each unclosed leaf individually.

The scope of the counterexample for a given leaf is every statement between the leaf and nearest preceding point where all knowledge about the program state, in the form of collected updates, is lost. We refer to these points as **full anonymization points**. As no information about the program state is kept across a full

anonymization point,[1] reconstructing anything beyond the closest anonymization point is meaningless as any state changes occurring before can be overwritten at the anonymization point.

For crowbar and ABS, full anonymization points are:

- The start of a method
- The loop invariant preserves case, as invariant preservation has to be shown for arbitrary program states
- The loop invariant use case, as the use-case can assume no more than the loop invariant

To generate a counterexample, we can therefore start at the leaf node in question and traverse the symbolic execution tree upward until we reach a full anonymization point. For each traversed node, we can use the information stored in the tree to reconstruct the statement whose symbolic execution resulted in this node.

This scope definition allows us to handle special cases like loop invariants in the same way as a simple method body.

When combined, these reconstructed statements form an excerpt of the original program that shows precisely the execution path causing the verification to fail. Any code blocks not relevant to the execution path — e.g. conditional branches not taken — are excluded from the counterexample by design.

3.2 Model Integration

With the execution path reconstructed, we now have to modify the obtained program to include information from the SMT model to make the counterexample exhibit specification-violating behaviour.

Recall that the SMT model is a set of assignment that satisfies the input formula and that said input formula is the negation of the formula we attempt to show. The assignments in the model thus represent a program state which satisfies all preconditions in which at least one postcondition does not hold. If we modify our counterexample to match this model program state, we can therefore reliably create undesired behaviour.

Concretely, these modifications have to occur in two different places: Before the code block in question, simulating the program state at the time the method or block is executed, and at any point where another process might change the program state. As other processes can not interfere with local variables defined inside a method, we introduce the notion of **heap anonymization points**. At a heap anonymization point, contents of the heap may be arbitrarily changed by another process within the bounds of invariants specified by the developer. In contrast to full anonymization points, knowledge about local variables is preserved across a heap anonymization point.

Just like for statement reconstruction, we augment the symbolic execution tree at these anonymization points with information allowing us to infer the correct program state changes at a given point from the SMT model.

[1] We are referring here to concrete updates gathered during the symbolic execution. The possible program states are still constrained by specification elements such as preconditions and invariants.

3.3 Context Removal

With the previous two steps completed, we are now able to create a counterexample that exhibits specification-breaking behaviour. However, this counterexample can still contain method calls and object creations that depend on external context and statements that can block during the program execution.

To obtain a counterexample that can be executed and analyzed with no external dependencies, we will re-write blocking or context-dependent statements to dummy-statements that are equivalent for the purposes of our counterexample.

We chose this approach of context removal over the possible alternative of integrating the generated counterexample into the context of the original method, which would have the advantage of generating counterexamples that are closer to the original code. We feel that the reduced size and complexity of dependency-free generated code makes it easier for the human verifier to understand the example and thus makes it more useful than a faithful reproduction of the original code.

3.4 Counterexample Generation

With the three main components of program reconstruction, model integration, and context removal laid out, we can now define our high-level method for counterexample generation. We present an incomplete pseudocode implementation for context in Listing 3.1.

Our algorithm takes as its parameters an annotated symbolic execution tree, which was previously augmented with all information necessary to reconstruct statements, and a leaf in said tree, belonging to the branch in the proof tree that was unable to be closed. We then traverse from this leaf upwards towards the root of the tree, collecting all nodes we encounter in a list, stopping once we reach a full anonymization point. The nodes collected in this list represent the execution path we will reconstruct — reaching from the unclosed leaf up to the nearest point of complete information loss.
We can now generate our counterexample by iterating over the list in reverse order and reconstructing one statement from each node in the list. Note that it is not strictly speaking necessary to go through the list in reverse order — however, reconstructing statements in the order they were executed makes it much easier to account for scoping and definition issues that we cannot reconstruct from the symbolic execution tree directly.[2]

The reconstruction and context removal of individual statements depends heavily on the given statement, which is why we do not provide a concrete method here. In general terms, the implementation of *renderAndContextRemove* has to assemble the information contained in the annotated tree node into a string representation of the statement that was executed, with possible modifications to remove context requirements or blocking statements. These modifications may also require the use of model information.[3]
Independent of the given original statement, the reconstructed statement will include assignments from the logic model if it is an anonymization point. For heap anonymization points, the anonymized heap expression from the tree node is used to obtain values for all fields in this state. For full anonymization points, additional assignments to parameters and local variables are inserted.

[2] These issues will be discussed in more detail in the implementation section.
[3] For example for replacing the evaluation of future types or, to a lesser extent, the creation of new objects.

Once every statement is reconstructed and rendered, we assemble the final counterexample by encasing the generated code in a predefined frame including the necessary package-, class-, and method definitions to obtain an executable program file. This frame also includes required field declarations and other class-level definitions.

```
1  GENERATE–CE(dst, leaf)
2    ce = ""
3    list = EmptyList
4    node = leaf
5    WHILE node ≠ null {
6      ADD node TO list
7      BREAK IF node IS FULL ANONYMIZATION POINT
8      node = node.parent
9    }
10   list = REVERSE(list)
11   FOR s ∈ list DO {
12     ce += RECONSTRUCT–STATEMENT(s)
13   }
14   RETURN frameHeader + ce + frameFooter
15
16 RECONSTRUCT–STATEMENT(node)
17   stmt = renderAndContextRemove(node)
18   IF node IS FULL ANONYMIZATION POINT {
19     stmt += varAssignments()
20   }
21   IF node IS ANY ANONYMIZATION POINT {
22     stmt += heapAssignments(node.postHeap)
23   }
24   RETURN stmt
```

Listing 3.1: High-level counterexample generation algorithm

4 Implementation

In the following, we will go over the concrete implementation of the previously described method for crowbar and ABS. Just like crowbar, our implementation will be written in the Kotlin language.[1] As the crowbar tool is still in the early stages of development and the counterexample generation functionality is largely separate from the tool's main functionality, we will try to keep the two parts as isolated as possible and make only minor modifications to the core architecture of crowbar. This should allow for modifications to both the core and counterexample components with little knowledge of the other.

4.1 Information Collection

To reconstruct executed statements, we first have to gather and save the required information during the symbolic execution process. Keeping in mind that we want to avoid major changes to the crowbar architecture, we encapsulate all gathered information in *information objects* and add references to such objects to the nodes of the symbolic execution tree. Concretely, we add the *info* property, referencing a *NodeInfo* object, to crowbar's *SymbolicNode* and *LogicNode* classes. While the former represents inner nodes of the execution tree where parts of the program still remain un-executed, the latter represents leaf nodes consisting only of logic formulas.

For each relevant type of node, we add a subclass of *NodeInfo* containing all required information. We distinguish types of nodes primarily by which rule of crowbar's calculus was applied to create it (i.e. the rule for executing an if-statement) as this will tell us which statement we need to reconstruct. Some rules, however, split the proof tree by introducing several new nodes such as if- and else-branches or null-checks. Thus, some rules are associated with multiple node types.

Aside from information for statement reconstruction, each *NodeInfo* object also identifies whether its associated node is a full anonymization point, a heap anonymization point, or neither of the two. We also distinguish what we call *significant branches* — branches in the proof tree that show obligations other than the method postcondition. Common significant branches are showing invariants, proving loop invariants, and checking that an object is non-null. Regular branches within the program, such as in the case of an if-statement, are not considered significant branches as both resulting paths in the proof tree attempt to show the method postcondition.

We provide a list of all subclasses of *NodeInfo* with the information they contain in Table 4.1. This listing includes information strictly necessary for program reconstruction (in bold) as well as information collected to provide auxiliary information.

[1] See `https://kotlinlang.org/`

Name	SigBranch?	Saved information
Invariant	y	object invariant
LoopInitial	y	**loop guard**, loop invariant
LoopPreserves	y	**loop guard**, loop invariant
ClassPrecondition	y	class precondition
MethodPrecondition	y	method precondition
NullCheck	y	non-null assertion
ScopeClose	n	*none*
AwaitUse	n	**guard expression**, anonymized heap expression
LoopUse	n	**guard expression**, **loop invariant**
IfThen	n	**guard expression**
IfElse	n	**guard expression**
LocAssign	n	**location**, **expression**
GetAssign	n	**location**, **expression**
CallAssign	n	**location**, **callee**, **call expression**, future identifier
SyncCallAssign	n	**location**, **callee**, **call expression**, anonymized heap expression, return value expression
ObjAlloc	n	**location**, **class init expression**, object identifier expression
Return	n	**return expression**, method postcondition, object invariant, current update
Skip	n	*none*
SkipEnd	n	method postcondition

Table 4.1: Subclasses of NodeInfo with their associated information, with entries essential for program reconstruction marked in bold.

4.2 Model Parsing

Apart from the annotated symbolic execution tree, the second important source of information we use for counterexample construction is the model provided by the SMT solver. Just like the instructions to the solver, the model is described in the SMT-LIBv2 language, which we will have to parse to access the relevant information.

While *z3* offers bindings for several programming languages, including Java,[2] which can be used to parse SMT-LIB inputs, using these bindings would come with two significant disadvantages: Firstly, using these bindings requires a z3 version that is compiled with a corresponding option enabled, meaning that common binary distributions of the solver might not work with our tool. Secondly, this would restrict the counterexample generation to z3, whereas the rest of crowbar is deliberately designed to work with any SMT-LIB compatible solver (e.g. cvc4), making it easy to use newer or more specialized tools when they become available.

The obvious alternative to using these bindings would be using an existing parser for the SMT-LIB language — and in fact, the jSMTLIB project [6] appears to provide a rather complete implementation of SMT-LIB in Java. The tools for parsing SMT-LIB inputs, however, seem to be completely undocumented at the time of writing.

But even if suitable SMT-LIB parsers for Java or Kotlin were available, extracting useful information from

[2] See `https://github.com/Z3Prover/z3/#z3-bindings`

the model would be much more difficult than simply traversing the parsed AST. The main reason for this is how crowbar models the anonymization of the program heap that occurs, for example, during method calls. When symbolically executing a method call, crowbar simulates possible changes to the heap by applying an anonymization function to the current heap. The SMT-LIB interpretation of this anonymization function usually looks like in the example given in Listing 4.1, with the complexity of the function increasing with the number of anonymization points in the method. Taking a look at the example, we can see that even if we know the exact state of the heap before the anonymization point, obtaining the modified heap would require evaluating several helper functions — `k!4` and `k!5` in the example model — and comparing them to the heap before anonymization as part of the nested `ite` (if-then-else) expressions. As the helper functions themselves can be non-trivial, this would essentially require a run-time interpreter for parts of the SMT-LIB language.

```
(model
  (define-fun heap () (Array Int Int)
    ((as const (Array Int Int)) 1))

  (define-fun anon ((x!0 (Array Int Int))) (Array Int Int)
    (ite (= x!0 (_ as-array k!4))
      (store (store ((as const (Array Int Int)) 5) 10 11) 8 9)
    (ite (= x!0 (_ as-array k!5)) (store ((as const (Array Int Int)) 6) 8 7)
      (store ((as const (Array Int Int)) 2) 10 12))))

  (define-fun k!4 ((x!0 Int)) Int
    (ite (= x!0 10) 12
    (ite (= x!0 8) 4
      2)))

  (define-fun k!5 ((x!0 Int)) Int
    (ite (= x!0 10) 11
    (ite (= x!0 8) 9
      5)))
  [...]
)
```

Listing 4.1: Excerpt from a model showing the solver's interpretation of the anonymization function

Because there are no obvious off-the-shelf solutions for our problem of parsing and evaluating SMT-LIB models and writing a complete parser for the language is a daunting project on its own, we will use a different approach to extract information from the models. While the SMT-LIB language in its entirety is very complex, the statements that appear in generated models are more constrained, and the most basic building blocks of such models — constant definitions of integer and array types — are very easy to parse. Apart from these simple definitions, the models can and do still contain more complex constructs such as function definitions. Luckily for us, however, the SMT-LIB language includes the *eval* command, which allows us to evaluate arbitrary SMT-LIB expressions, returning simple integer or array definitions.

These commands can be appended to the same call to the SMT solver in which we obtain the model in the first place, allowing us to evaluate arbitrary expressions in the model context while causing negligible overhead. As crowbar already provides methods for translating expressions into their SMT-LIB representation, we can trivially collect expressions for evaluation while gathering information during symbolic execution.

In conclusion, we extract information from SMT-LIB models by using a simple recursive-descent parser that understands definitions of integer and integer-array values while ignoring most other constructs of the SMT-LIB

language. For the constructs of interest that are too complex for this parser, we use the *eval* command of SMT-LIB to reduce the constructs to simple values that we *can* parse.

Concretely, we are interested in the following kinds of information found in the models:

1. Initial states of method parameters and variables
2. Initial state of the heap
3. Heap-states after heap anonymization points
4. Future evaluations (i.e. the value that a future will resolve to)
5. Various expressions for additional information (e.g. return expression evaluation)

With this in mind, let us now take a closer look at the anatomy of a typical model as presented in Listing 4.2 to see how we can obtain the required information from it.

The first line in the solver's output represents the satisfiability of the input problem: An unsatisfiable input corresponds to a successful proof for which no counterexample is necessary or even possible. For a satisfiable input, we can generate counterexamples as described in the following. The third possible result here is *unknown*, which indicates a timeout in the solver before a definite solution was reached. In this case, model data is unavailable and all information derived from the model will be marked as unknown in the generated counterexample.

Next in the output is the model itself from line 2 to 14. This model contains the initial state of the heap (lines 3–4), which is modelled as an infinite array of integer values, initialized in this case to 4 at every position. Note that the model operates entirely on integers and has no concept of fields, objects and futures. This means that we have to translate these concepts into their integer representations and back — for fields, we can obtain this mapping from the definitions in lines 5 and 6, where the heap-indices for fields *f* and *g* are defined.

In line 7, the method parameter *v* — essentially a local variable — is initialized to 0. The following two lines represent the mapping of object identifiers, specifically those created by *new* expressions, to their integer counterparts. While the two lines in this model define easily-parseable constants, they can also be instantiated with parameter-dependent functions in more complex models, which is why we will use *eval* commands to evaluate new object expressions instead.
The last line of the main model gives us the mapping of future-identifiers to their integer representations. Note that these are not the value that a future will resolve to, but simply the integer representation of the future itself for the SMT model. The resolved value of a future is modelled by the *valueOf* function, which can be applied to the future's SMT representation.

Following the main model are the results of the *eval* commands issued by the crowbar investigator. There are three *eval* commands, each of which can evaluate many expressions. The first command (line 15) evaluates anonymized heap expressions. In this example, the heap after anonymization is an array of constant value 6 with the values 8 and 7 inserted at positions 1 and 0 respectively. With knowledge of the mapping from lines 5–6, this translates to the assignments `f = 7` and `g = 8`.
Line 16 shows the command used to evaluate new object expressions. If any of the `NEW` definitions were functions rather than constants, their parameters would be included here.
Finally, the last line of the output evaluates various other expressions. The first one in this example is a return-expression which, as the method in question appears to always return 1, rather trivially evaluates to 1. The second expression is used to obtain the resolved value of a future stored in field *g*, which is 0.

```
sat
(model
  (define-fun heap () (Array Int Int)
    ((as const (Array Int Int)) 4))
  (define-fun f_f () Int 0)
  (define-fun g_f () Int 1)
  (define-fun v () Int 0)

  (define-fun NEW0_0 () Int 3)
  (define-fun NEW1_0 () Int 7)

  (define-fun fut_3 () Int 5)
  [...]
)
(((anon (store (store heap f_f NEW0_0) g_f fut_3)) (store (store ((as const (Array Int Int)
    ) 6) 1 8) 0 7)))
((NEW1_0 7) (NEW0_0 3))
((1 1) ((valueOf (select (anon [heap]) g_f)) 0))
```

Listing 4.2: Excerpt from a typical SMT solver output, including satisfiability result, model, and expression evaluations. Square brackets denote modifications for brevity.

We store this extracted information in a *model object*. This object contains the initial program state, a mapping of heap expressions collected from anonymization points to their corresponding heap states,[3] lookup tables mapping model values to the futures and objects they represent, and a general-purpose mapping of SMT-LIB expressions to their evaluations. This general-purpose mapping is used for future evaluation and various other expressions that need to be evaluated. Expressions of interest can be defined in each NodeInfo object. During counterexample generation, expressions are collected from every NodeInfo object, evaluated, and added to the model object.

One problem that we encounter using this approach is that of missing identifier definitions: When collecting SMT-LIB expressions for evaluation during symbolic execution, we might collect an expression containing a variable that is not relevant to the current proof, i.e. it occurs neither in the precondition nor in the postcondition of the *LogicNode* we are investigating. This can occur, for example, when obtaining the value of a future-type local variable that is not used anywhere else. Since the variable declarations that are input into the SMT solver are generated from the pre- and postconditions of a LogicNode, attempting to reference an identifier that is not present in either of these will cause the SMT solver to produce an error.
Fortunately, any expression that would cause this error is irrelevant to the proof per definition since it has no effect on pre- or postcondition. As such, it is of no interest for our counterexample.[4] We can, therefore, simply check every expression for problematic identifiers before passing them to the SMT solver and omit any expressions that contain them. We will render these unevaluated expressions as irrelevant or unknown values in the generated counterexamples.

[3] The parsed representation of heap states is already in the form of a 'field to value' mapping, which means we do not have to remember the 'field to heap index' mapping.
[4] This behaviour could possibly be exploited for counterexample minimization, as we will point out in our conclusion.

4.3 Statement Reconstruction

To recreate the ABS code that was symbolically executed on a given path, we will apply a reconstruction or *rendering* function to each NodeInfo object on the path. This function renders the information contained in the info object to a string representation of the corresponding ABS statement.
As previously stated, we attempt to keep counterexample generation and normal verification logic as separate as possible. To implement a rendering function for every type of NodeInfo object, we use the *visitor pattern* [10, pp. 366–380], allowing us to contain the entire reconstruction logic in a single class which we will call *NodeInfoRenderer*.

Besides pure statement reconstruction, these rendering functions also apply expression reconstruction, context removal, and model integration in the same step of the counterexample generation process. We will discuss each of these aspects in the following sections.

The statement reconstruction itself is, as previously noted, rather trivial and highly dependent on the concrete statement in question. For this reason, we won't discuss it here in detail.[5] There are, however, two interesting problems that arise because crowbar's translation from ABS into its internal representation is not entirely lossless:

Firstly, crowbar does not distinguish variable declarations from variable assignments because it is not necessary to declare identifiers before use in the underlying formalization. As a consequence, we have to insert declarations in the reconstructed statements where necessary to create compilable ABS code.
As ABS does not allow variable shadowing [1, sec. 6.2], in which a variable in an inner scope prevents access to an outer variable of the same name, or variable re-declaration [1, sec. 6.2], we can simply change the first assignment to an identifier to be a declaration instead. Iterating over the information objects in source-code order, we can add an initially empty list of identifiers to the *NodeInfoRenderer* class. When an assignment is encountered, we check this list for the identifier on the left-hand side of the assignment. If there is no valid declaration for this identifier in the current scope, we add the identifier to the list and render a declaration instead of an assignment. A pseudocode implementation of our assignment rendering process, which also handles the second problem discussed below, can be found in Listing 4.3.

Secondly, and for similar reasons, crowbar ignores scoping. Primarily, this means that we do not know where to end reconstructed branch or loop bodies, but it also poses some more subtle issues for the declaration reconstruction outlined above.
While these issues could be solved by explicitly modelling scoping with a new set of rules, this would add four or more rules to crowbar's calculus and require major changes to the existing 11 rules. This, we feel, would lead to a disproportionate increase in complexity of the crowbar system compared to a minor improvement in counterexample generation.

Instead, we will address the first problem by introducing a new program element to crowbar's internal representation of ABS code, together with a single new rule in the calculus. The new program element, which we will call the *ScopeCloseMarker*, essentially acts as a special kind of *skip* statement. When symbolically executing a conditional or a loop statement, this statement is inserted after the statements from the conditional or loop body and before the remaining code. The corresponding rule we introduce, presented in Figure 4.4, is an exact copy of the *skip* rule, with the exception that it matches *ScopeCloseMarker* statements rather than *Skip* statements. The soundness of this new rule follows directly from the soundness of the *skip* rule. As we

[5] For more details on statement reconstruction, please refer to our implementation at `https://github.com/rec0de/crowbar-tool/blob/master/src/main/kotlin/org/abs_models/crowbar/investigator/NodeInfoRenderer.kt`.

```
definedVariables IS LIST OF (identifier, scope)
declaredTypes IS MAP OF identifier -> type

RENDER-LOC-ASSIGN(infoObj)
  location = infoObj.location
  expression = infoObj.expression

  // Declare variable on first use
  IF location ∉ definedVariables
    ADD (location, currentScope) TO definedVariables
    type = TYPE OF expression
    SET declaredTypes[location] TO type
    renderedLocation = type + " " + location
  ELSE IF (location ∈ definedVariables) ∧ (declaredTypes[location] ≠ TYPE OF expression)
    // Variable re-declaration of different type due to lost scoping
    type = TYPE OF infoObj.expression
    disambName = location + "_redec" + type
    renderedLocation = type + " " + disambName

    // Re-map all future uses of location to disambName

    // Declare renamed variable & type
    IF disambName ∉ definedVariables
      ADD (disambName, currentScope) TO definedVariables
      SET declaredTypes[disambName] TO type
  ELSE
    // Variable is already declared
    renderedLocation = location

  RETURN renderedLocation + " = " + expression + ";"
```

Listing 4.3: Pseudocode implementation of our variable declaration and assignment reconstruction. For brevity, it is assumed that all assignment-locations are local variables. The *currentScope* variable gives the depth of the current scope as an integer. When a scope is closed, all entries in *definedVariables* of that scope are cleared.

only introduce ScopeCloseMarker statements at points where they are followed by other statements,[6] the remaining program s' will never be empty and all added ScopeCloseMarker statements can be executed by our rule. If we now look at the *NodeInfo* objects along a given execution path, we will find an info object created when executing the ScopeCloseMarker, from which we can reconstruct the correct point to close the current scope.

$$\frac{\Gamma \Rightarrow \{U\}[s' \Vdash L], \Delta}{\Gamma \Rightarrow \{U\}[\text{ScopeCloseMarker};s' \Vdash L], \Delta}$$

Figure 4.4: Rule for symbolic execution of the ScopeCloseMarker statement.

With this tweak, we can now faithfully reconstruct scopes attached to conditionals or loops. Stand-alone scopes, however, are still ignored. For the most part, this is not an issue — as mentioned above, variable

[6] Even if the program ends after a scope close, such as in the case of loop bodies, the ScopeCloseMarker will be followed by a *skip* statement.

shadowing or redeclarations are forbidden in ABS, meaning that removing some scoping from the program will only lead to semantic changes in a few limited edge-cases. These are illustrated in Listing 4.5: The variable *a* is declared twice with the same type — the first declaration is only valid until the end of the scope in line 4, meaning that the same identifier can be re-declared in line 6. Our reconstruction approach already handles this edge case, since we infer declaration points as described above. If we drop the scoping around lines 1–4, the second declaration would simply be rendered as an assignment to the existing variable. More difficult is the second case, in which *b* is re-declared with a different type. Our current approach would incorrectly render line 7 as an assignment of 42 to a boolean-type variable.

```
1 {
2   Int a = 10;
3   Bool b = True;
4 }
5
6 Int a = 10;
7 Int b = 42;
```

Listing 4.5: Illustration of problems arising from lost scoping and identifier reuse in ABS.

Following the intuition that the latter edge case is sufficiently rare in real-world programs as it requires both a stand-alone scope block and the redeclaration of an identifier with a different type, we opt for a simple detection-based approach to this issue. When rendering an assignment, we check the type of the assignment against the declared type of the variable. If the types do not match, we have detected a re-declaration. In this case, we include a warning in the counterexample, declare a renamed variable of the appropriate type, and replace all future uses of the original variable with the renamed one. This approach is also outlined in Listing 4.3.

4.4 Expression Reconstruction

Just like we need to render statements back into valid ABS code, we also have to render the expressions that are part of these statements. As ABS expressions are, with few exceptions that are handled explicitly, side-effect free[1, sec. 4], crowbar does not execute them like statements but converts them into equivalent formulas in a single step.[7] Unfortunately, this conversion is not as lossless as the statement conversion: The expressions *this*, *1*, and *True*, for example, are all converted to *1*, while *False*, *null*, and *0* are converted to just *0*. Binary operators are also re-written as function applications in prefix notation, making a potential conversion back to ABS code quite tedious.

To accurately render expressions in the way they are written in the source code, we therefore opt to save a reference to the original ABS expression. There are two obvious places to keep these references: Saving them in the statement objects would result in the least memory-overhead since references are only saved for top-level expressions. However, changing the structure of crowbar's internal program representation in this way breaks the pattern matching used to find applicable rules during symbolic execution. While this problem can certainly be solved by adjusting the matching algorithm, doing so would require significant changes to a core part of the crowbar tool. The second, more flexible option is to store references to original

[7] Pure ABS expressions can technically still cause exceptions (e.g. for a division by zero), which the single-step translation does not account for. This is not an issue, however, as crowbar currently only supports specification of normal behaviour, i.e. executions of the program that terminate normally.

ABS expressions in crowbar's internal expression objects. This allows us to convert arbitrary sub-expressions back to their original ABS form at the cost of a slight increase in memory used per expression object.

Considering its flexibility and simplicity, we choose the latter option for our implementation. Concretely, we add the field *absExp* to all crowbar-internal expression classes. This field is then initialized with the appropriate reference during the conversion of ABS expressions.

4.5 Context Removal and Model Integration

With model parsing and program reconstruction in place, the only remaining steps in our counterexample generation method are context removal and model integration. As we will see, these two form a very natural union as we replace statements requiring external context with new statements that model their behaviour. Also, inserting assumptions from the model at all the places within the program where unknown state changes can occur will ensure that the counterexample we generate exhibits the specification-breaking behaviour that the model encodes.

There are two groups of statements that we will need to modify: Statements involving references to objects, as the rendered counterexample will lack any external class definitions, and statements involving futures, as the counterexamples will not perform any method calls or wait for values to become available. For both objects and futures, we have to find replacements that are compatible with all the ways that the originals can be used.

Object references, for example, can be used in just two ways besides simple assignments in the subset of ABS supported by crowbar: We can use them in an expression to check for referential equality or we can use them to synchronously or asynchronously call a method on the object. If we simply replace all object creations with string literals and adjust variable and field types accordingly, we can model referential equality checks by comparing the string literals instead as shown in Listing 4.6. Using strings in particular comes with the added benefit that the created literals are easy to understand for human readers.

```
1  // Before context removal
2  SomeInterface a = new SomeInterfaceImplementation();
3  SomeInterface b = a;
4
5  if(a == b) {
6    // Do something
7  }
8
9  // After context removal
10 String a = "object-1";
11 String b = a;
12
13 if(a == b) {
14   // Do something
15 }
```

Listing 4.6: Example of context removal for object references and referential equality.

To assign unique string literals to each object, we can make use of the fact that the SMT model already assigns integer IDs to each object which are easily converted into string literals.[8]

[8]While not strictly necessary, we perform some re-mapping of model IDs to string literal IDs for more consistent object numbering.

With object references now replaced by strings, we have to remove all method calls from the counterexample since a method call on a string variable would make for an invalid ABS program. For synchronous method calls, we can replace the call with a simple assignment to the target location. We obtain the value for this assignment from the SMT model, where we have previously evaluated the modelled return value of the call (see 4.2). While the value obtained from the model is a simple integer value, we can use the type information from the original call to render the value in the correct type as shown in Listing 4.7. If, for example, the return value is an object reference, we can interpret the modelled return value as a model object ID and render the corresponding string literal. If the return value of the method call is not assigned to any target location, we can simply omit the dummy assignment.

```
1 RENDER-MODEL-VALUE(value, type)
2   WHEN type IS
3     Integer: RETURN toString(value)
4     Future: RETURN "future-" + toString(value)
5     Boolean: RETURN "False" IF value = 0, ELSE "True"
6     ELSE: RETURN "null" IF value = 0, ELSE "object-" + toString(value)
```

Listing 4.7: Pseudocode implementation of model value rendering. Slightly simplified to not include future and object identifier remappings or unknown types.

We also have to consider that the called method might modify any value on the heap.[9], making synchronous calls *heap anonymization points*. To account for possible changes, we include assignments to each heap location in our replacement that bring the heap into the state of the anonymized heap we obtained from the model. Again, we convert integer model values to the appropriate types using the technique described above.

Asynchronous calls, on the other hand, return immediately and do not allow for any heap modifications. We can therefore replace them with a simple assignment. For the future that is returned by the call, we will proceed as we did for object references: We replace the future with a string literal and change all future types to string types. Again, we can base the generation of string literals on the integer IDs available in the model.

Examples for both types of calls are shown in Listing 4.8.

```
1  // Before context removal
2  Int a = obj.m();
3
4  Fut<Int> b = obj!m();
5
6  // After context removal
7  this.field1 = 0;
8  this.field2 = False;
9  Int a = 42;
10
11 String b = "future-1";
```

Listing 4.8: Example of context removal for synchronous and asynchronous method calls. Lines 7–8 show heap anonymization.

Futures can be used in three different ways: Just like object references, we can check them for equality. Additionally, we can obtain the resolved value of a future in a *get* expression or wait for the future to resolve

[9]While the synchronously called method might be in a different object within the same cog which does not have access to the current object's heap, this method could in turn call a method in the current object and thereby modify the state.

using an *await* statement. As for object references, the string literal replacements already support equality checks without further modifications.

As *await* statements only block the program execution until the future resolved, we can remove them without changing the semantics of our counterexample. However, as the *await* statement releases control of the processing unit [1, sec. 6.6], another process on the same COG might be scheduled while the *await* statement blocks execution. Because such a process could modify values on the heap, all *await* statements are *heap anonymization points*. As for synchronous calls, we have to include an assignment block modelling this heap anonymization.

Just like the *await* statement, the *get* expression blocks execution until the future in question resolves. Unlike the *await* statement, however, it does not release control of the processing unit [1, sec. 4.2], which means no changes to the local heap are possible. We can therefore replace the expression with the resolved value of the future, which we have already obtained from the model as described in 4.2.

Context removal for both *await* and *get* is illustrated in Listing 4.9.

```
1  // Before context removal
2  Fut<Int> a = obj!m();
3
4  await a?;
5
6  Int b = a.get;
7
8  // After context removal
9  String a = "future-1";
10
11 // Heap anonymization
12 this.field1 = 0;
13 this.field2 = False;
14
15 Int b = 42;
```

Listing 4.9: Example of context removal for *await* statements and *get* expressions.

Finally, we also replace *return* statements with equivalent *println* statements. This allows us to use a static method frame with unit return type for all counterexamples while still providing relevant information during execution.

It is also worth noting that branches or loops do not require any context removal since the modelled assignments already ensure that guard expressions evaluate to the correct values (i.e. such that a loop is not entered or the branch that was not reconstructed is not taken).

4.6 Counterexample Generation

We now have all the necessary prerequisites to generate counterexamples from symbolic execution paths. All that remains now is to add some minimal context around the generated code to turn it into valid ABS. To do so, we embed the counterexample statements we obtained by applying our rendering-function to every NodeInfo object on the symbolic execution path into a predefined frame that is shown in Listing 4.10.
As outlined in the abstract description of our method in 3.4, we always reconstruct the executed code up

to the closest full anonymization point — usually the beginning of the method or an application of a loop rule. As such, we have to include assignments modelling the initial state in which the counterexample is to be executed. To model the initial heap state, we add declarations of all used fields to the class declaration in the frame that initialize the fields to their initial values or a default value if initial values are unavailable. These field declarations also reflect the type replacements discussed in 4.5 — replacing future and interface types with string types for compatibility with the counterexample code. For method parameters and local variables, we include an assignment block at the start of the counterexample method that assigns the correct values.

As our approach assigns model values for method parameters in the form of local variables, the counterexample method in our frame does not need any parameters. Likewise, we can use a unit return type for the counterexample method as we remove any return statements during context removal.

To summarize, we generate a counterexample using the following steps in the given order:

1. Extracting an unclosed leaf and its corresponding execution path from the symbolic execution tree
2. Obtaining and parsing model information
3. Rendering initial assignments to local variables found in the model
4. Rendering each statement on the execution path
5. Rendering declarations of all used fields
6. Combining initial assignments, rendered statements and field declarations into the counterexample frame

The order of these steps is important for two reasons: We have to render the initial assignments before rendering the statements themselves so that variables declared in the initial assignments are properly recognized as already declared when rendering statements. Also, it is advisable to render field declarations *after* rendering statements as it allows us to collect all fields used in the counterexample during statement rendering. In some cases, not all fields included in the counterexample are relevant to the given proof, and thus might not be included in the SMT model. Using the fields actually used in the counterexample as the source for field declarations ensures we produce valid ABS code.

In our implementation, this process is automatically repeated for every unclosed leaf in the tree, producing multiple separate counterexamples.

This is also a good time to point out that, while we have not accounted for these cases explicitly, our method already handles loops and other *significant branches*, i.e. branches in the tree that attempt to show something other than the main postcondition, gracefully. Executing a loop statement, for example, splits the execution tree into three parts. The first one attempts to show that the loop invariant holds initially. A counterexample for this branch would reconstruct the statements up to the loop statement with assignments such that the loop invariant does not hold. The second branch attempts to show that a single execution of the loop body preserves the invariant, given it holds initially. The first node in this branch is a full anonymization point as this proof assumes the loop body is executed in an unknown state. Thus, the counterexample generated would consist of the loop body reconstructed up to the loop statement, with assignments such that the initial state satisfies the invariant but the state after execution does not. The third and final branch, then, shows that the execution of the remaining code after the loop satisfies the specification, given that the loop invariant holds after executing the loop. The first node on this branch, again, is a full anonymization point, and the corresponding counterexample will include the remaining code of the method up to the end of the loop body.

```
module Counterexample;

interface Ce { Unit ce(); }

class CeFrame implements Ce {
  // field declarations inserted here

  Unit ce() {
    // counterexample inserted here
  }
}

{
  Ce x = new CeFrame();
  x.ce();
}
```

Listing 4.10: The predefined frame for all generated counterexamples.

5 Enhancing Clarity

In our problem definition, we included clarity and abstraction as explicit goals for our counterexample generation method. So far, we have focused on the technical aspects of generating correct counterexamples without much regard for either clarity or abstraction.

The context removal process, for example, can easily create ambiguity as to what the function of a given statement was in the original program as information about method calls and await conditions is lost entirely. And while the counterexample hides a large part of the proof complexity, the user still has to have enough knowledge of the underlying system to understand what it was trying to show at a given point.

To address these issues, we include the following additional information intended for human users in the form of comments in the counterexamples:

- Original statements wherever statements are removed for context removal
- Known-true premises for different stages of loop execution
- Evaluations of return expressions and contained identifiers
- Proof obligations in a given branch
- Sub-obligations that could not be shown

We will illustrate all of these using the ABS program shown in Listing 5.1 and its corresponding counterexample shown in Listing 5.2.

Looking at the counterexample, we can see that a copy of any statement that is replaced during context removal is kept as a comment (line 19). This is a trivial change to our rendering algorithm that makes it much easier to follow the counterexample and compare it to the original method.

In line 14, we can see that we are in a state after the loop has finished executing. We indicate this by including the loop statement with an empty body. Due to the initial assignments (lines 9–11) that set up a state in which the loop guard evaluated to false, this loop will never be executed. Because loop verification is one of the more complex parts of the proofs generated by crowbar, we include additional comments (lines 15–17) that remind the reader of what is known to hold in the state after loop execution, namely the loop invariant and the negated loop guard. While these constraints are already taken into account by the SMT solver when generating a model, increasing their visibility can help human users better follow the reasoning of the proof. We include similar reminders for proof branches showing that a loop invariant is preserved. All necessary information for these annotations is gathered during symbolic execution as part of the information collection process.

Following the return statement in line 22, we include the evaluated result of the return expression. This value could also be obtained by actually executing the counterexample, but with the model information we already have available, we can embed this information directly into the counterexample with very little extra effort.

```
1  module Test;
2
3  data Spec = ObjInv(Bool) | Ensures(Bool) | WhileInv(Bool);
4
5  [Spec : ObjInv(f == 42)]
6  class C {
7      Int f = 0;
8
9      [Spec : Ensures(result == 0 && result != 50)]
10     Int m(Fut<Int> v){
11         Int a = 10;
12         Int b = -5;
13
14         [Spec: WhileInv(f == 42)]
15         while(b < 0) {
16           Int tmp = a;
17           a = b;
18           b = tmp;
19         }
20
21         Int c = v.get;
22
23         return this.f - a + c;
24     }
25 }
```

Listing 5.1: Example program showcasing the different types of added information.

We have mentioned previously (see 4.2) that arbitrary expressions can be registered to be evaluated in the SMT model by adding them to a list of expressions in any NodeInfo object. Thus, we can simply add the return expression to the expressions of interest defined in the return statement's NodeInfo object and then access the evaluation of said expression, should one be available, when rendering the statement.
We can enhance this return expression evaluation further by including separate evaluations for any identifiers (i.e. fields or variables) used in the return expression. To do so, we extract any identifiers used in the return expression and add them to the same list to be evaluated. The integer model values are rendered according to the field or variable type as described in Listing 4.7.
This detailed evaluation, rendered as seen in lines 25 to 28, can be used to quickly determine which part of a complex return expression causes unwanted behaviour.

With the previous adjustments mostly aiming to improve clarity, we attempt to reduce the amount of knowledge of the internal proof system necessary to debug failing proofs with two additions to the counterexamples:

We explicitly include information about what the failing proof was attempting to show. This includes both the actual proof obligation as well as its type — such as *method postcondition*, *loop invariant preservation* or *object invariant*. We collect this information during symbolic execution by introducing a new subtype of NodeInfo objects: The *LeafInfo* class contains additional information about proof obligations and is used in place of a regular NodeInfo object for all leaves of the symbolic execution tree. When a LeafInfo object is encountered during the counterexample rendering, its proof obligations are embedded into the counterexample as seen in lines 29–31.
These annotations help orient the user within the proof by making clear which branch of the proof tree the counterexample represents.

```
module Counterexample;
interface Ce { Unit ce(); }

class CeFrame implements Ce {
  Int f = 42;

  Unit ce() {
    // Assume the following pre-state:
    String v = "fut_?(1)";
    Int a = 0;
    Int b = 0;
    // End of setup

    while((b < 0)){}
    // Known true:
    // Negated loop guard: !((b < 0))
    // Loop invariant: select(heap, this.f_f)=42

    // Int c = (v).get;
    Int c = 0;

    println(toString(((this.f - a) + c))); // return statement
    // Evaluates to: 42

    // Detailed evaluation breakdown:
    // this.f: 42
    // a: 0
    // c: 0
    // Proof failed here. Trying to show:
    // Method postcondition: (result=0) /\ (!result=50)
    // Object invariant: select(heap, this.f_f) = 42
    // Failed to show the following sub-obligations:
    // select(heap, this.f_f) - a + valueOf(v) = 0
  }
}
```

Listing 5.2: Output of the counterexample generation for the program in Listing 5.1.

As proof obligations can get quite long and complex in real-world programs, we also perform some additional analysis on them to identify which part of an obligation causes the proof to fail. The final formula that is constructed by crowbar and input to the SMT solver is of the form $\phi \implies \psi$, where ϕ is a precondition made up of information from the specification and symbolic execution. ψ, then, is the postcondition — a conjunction of all proof obligations like invariants and method postconditions. These obligations themselves often consist of a conjunction of smaller obligations — e.g. multiple statements about the contents of memory locations. Here, we can save human readers significant time and effort by automatically determining which of those conjunctively joined parts contributed to the proof failure, effectively reducing the search-space for debugging. One possible approach to this problem would be to modify ψ to exclude certain parts and re-run the proof, checking if it succeeds. Utilizing the infrastructure we already have available, however, we can implement another approach that does not require any extra invocations of the SMT solver. Concretely, we can extract all conjunctively joined sub-formulas from ψ before obtaining the model from the SMT solver. We can then include the extracted sub-formulas in the input to the solver to have them evaluated in our model. When parsing the model, we receive a boolean value for every sub-formula, indicating if it holds in the generated model. We can use this value to filter out only the sub-obligations that did not hold in the current model,

which we include at the end of the counterexample.
In line 33 of our example, we can see that this listing does not include the object invariant or the part of the postcondition specifying that the method may not return 50 because both of these conditions were met in the generated model. This has the added benefit that sub-obligations that are met in the current model but do not hold universally (result $\neq 50$, for example, could be violated in another model) are not included in the counterexample. This allows users to focus on the issue highlighted in the current example — once this issue has been addressed, iteratively repeating the verification process will yield another counterexample highlighting any remaining problem.

In summary, we hope to reduce the knowledge of the underlying proof system necessary to understand the reasons behind a proof failure by including information about currently relevant parts of proof obligations at appropriate points in the counterexample. Following a similar approach for enhancing clarity, we include detailed evaluations of potentially complex expressions to simple values at relevant points and preserve the original program structure to provide more context to a human reader.

6 Evaluation

In this chapter, we will evaluate our counterexample generation method with regards to the goals outlined in the problem definition. Recall that we defined these four main goals for our method:

- Small size
- Few dependencies
- High abstraction
- High clarity

Regarding dependencies, we have seen that we can completely remove external dependencies such as class or method definitions from counterexamples by applying context removal (see section 4.5). As such, we consider the goal of minimal dependencies to be fully met. We will investigate the three remaining goals in the following.

6.1 Counterexample Size

In order to be easily understood and analyzed by humans, generated counterexamples should be as small as possible, ideally reducing the amount of code to the bare minimum required to understand the issue at hand.

In our method, we achieve a reduction in size primarily by excluding branches not relevant to the current proof attempt and only reconstructing code up to the nearest anonymization point. Context removal also contributes to a lower size and complexity by replacing potentially large expressions with simple replacements.

To understand how these factors affect the size of real-world counterexamples, we have to test our method on a large variety of different inputs. Unfortunately, there is no sufficiently large corpus of ABS programs available at this time — especially not for programs restricted to the subset of ABS currently supported by crowbar. We therefore resort to randomly generating a large set of ABS programs.

6.1.1 Generating ABS

We build upon some of our previous work towards automatically generating valid *MAVL*[1] programs [22]. Extending and modifying the existing codebase, we create *MockABS*,[2] a script for generating random valid

[1] MAVL is short for the *Matrix and Vector Language*, an academic language used primarily to teach concepts of compilers at TU Darmstadt.

[2] The MockABS source code is available at `https://github.com/rec0de/crowbar-investigator-experiments/blob/master/mockABS.rb`.

ABS programs compatible with the current limitations of crowbar. MockABS starts with a static class template declaring fields of various types and populates it with a random implementation of a method with an integer return value. The template includes specification stating that the return value of said method is always zero. As the method body is entirely random, this claim will be false in the overwhelming majority of cases and will therefore lead to failing proofs.

As we want to make some claims about the size of real-world counterexamples in relation to the input method size, we have to ensure that our random programs resemble real-world code. This comes down to tuning three parameters of the MockABS script:

- Branch rate, the fraction of statements in the program that are branches (i.e. if-statements, loops, method calls)
- Statements per method, the average number of statements in a method body
- Statements per block, the average number of statements in a code block that is not a method body (i.e. loop bodies, blocks associated with conditional statements)

The statement per method and block parameters only consider the top-level statements in their scope. An *if* statement, for example, is counted as only one statement even if there are many statements in the associated *if* and *else* blocks. Statement counts are sampled randomly from an exponential distribution with the chosen expected value.
For the branch rate, we do not consider asynchronous method calls to be branches as they do not release control of the processing unit and are treated in the same way as other expressions by crowbar. Instead, we consider the *await* statement to be a branch for our purposes.

For the branch rate parameter setting, we take cues from research on compiler benchmarks by Nemer et al., which mentions common branch rates in the range of roughly 10–20 percent [21]. While these values can not be directly transferred to ABS and our notion of branches and statements, they do provide a rough estimate for sensible real-world branch rates. Due to lacking code statistics for ABS, we tune the remaining two parameters manually to produce reasonable outputs. We settle on a branch rate of 0.2, 10 average statements per method body and 4 average statements per code block. As we find that very large inputs take too long to verify, we also limit the total program size to 3500 bytes.

6.1.2 Size Analysis

For our analysis, we let MockABS generate 300 ABS programs using the parameter settings from above and attempt to verify each generated method using crowbar.[3] As there can be multiple uncloseable leaves in the proof tree for a generated program, one input program usually results in multiple — anywhere from 1 to 52 — counterexamples. In total, we obtain 945 counterexamples for the 300 input programs.

To compare program and counterexample sizes, we use two different metrics: The program's size in lines of code (*loc*) and significant lines of code (*sloc*). Both *loc* and *sloc* exclude empty lines, however comment-lines are included in the *loc* metric and excluded in the *sloc* metric. The intuition behind this distinction is that especially for small input sizes, the counterexample generation adds a relatively large amount of additional

[3] Unsurprisingly, testing our method on 300 random inputs is a great way to find unexpected behaviour. In the process, we find several edge-cases in our implementation as well as an incorrect rewrite rule in the ABS compiler (see `https://github.com/abstools/abstools/issues/283`).

information in the form of comments, which we feel does not reflect an increase in the complexity of the counterexample. The *sloc* metric might therefore better reflect true program complexity.

We present plots of input size vs. counterexample size for both metrics in Figures 6.1 and 6.2.

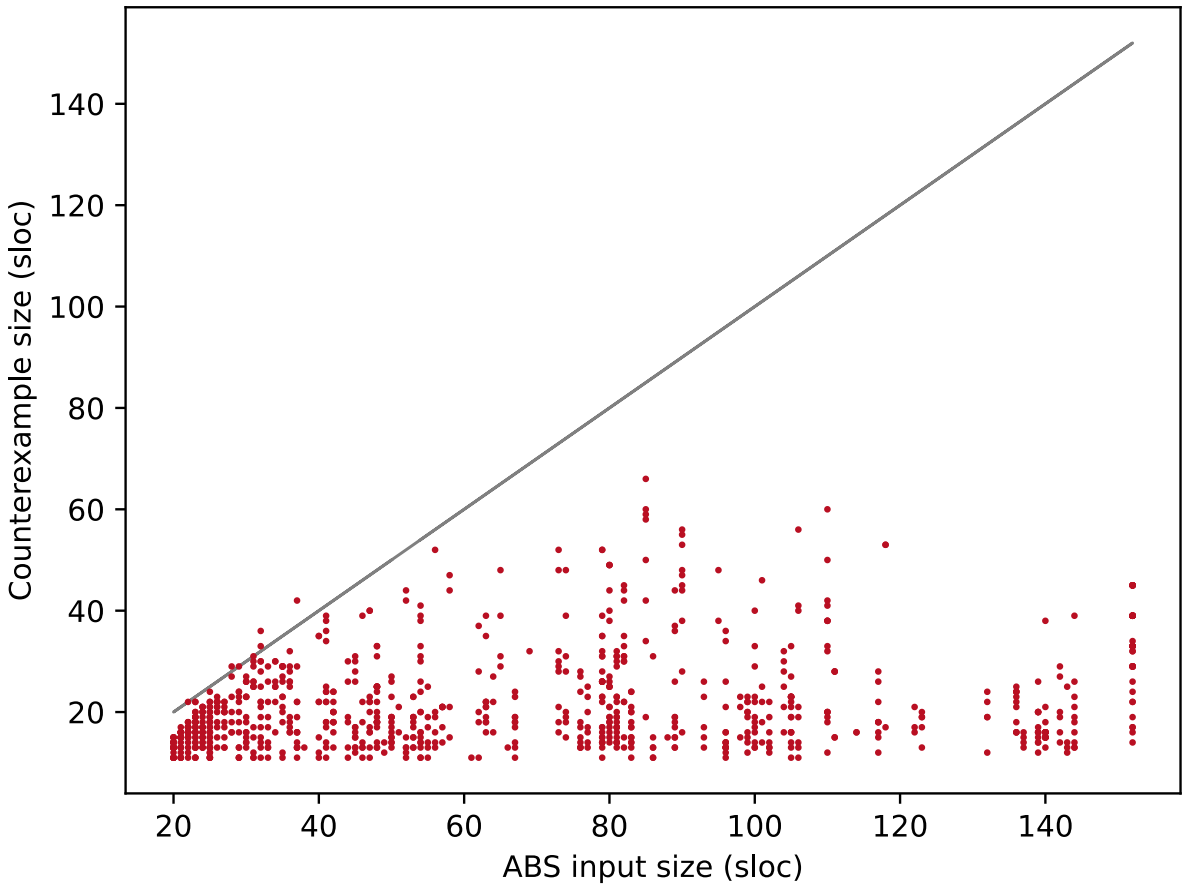

Figure 6.1: Input program size vs. counterexample size in significant lines of code for 300 randomly generated inputs. The diagonal shows the line of equal input/output size.

Both plots show that counterexample size increases linearly for small inputs but levels off to roughly constant levels for large inputs. The *sloc* measurements in particularly show that the counterexample complexity is lower than the input complexity for inputs of all sizes (see Figure 6.1). All generated counterexamples were below 70 sloc, with a median of only 18 sloc, including class-, field-, and method definitions.

In an attempt to confirm the results from this experiment — especially that counterexample sizes remain stable even for large inputs — we generate a secondary dataset of 100 ABS programs. We restrict this dataset to programs with 130 loc or more that are smaller than 5000 bytes. Because we have previously seen that the verification of very large programs sometimes stalls for long times, we adjust internal *MockABS* parameters to generate shorter expressions. As expressions are translated by crowbar in a single step and are not relevant to the symbolic execution itself, this should decrease the computational load of verification without affecting the

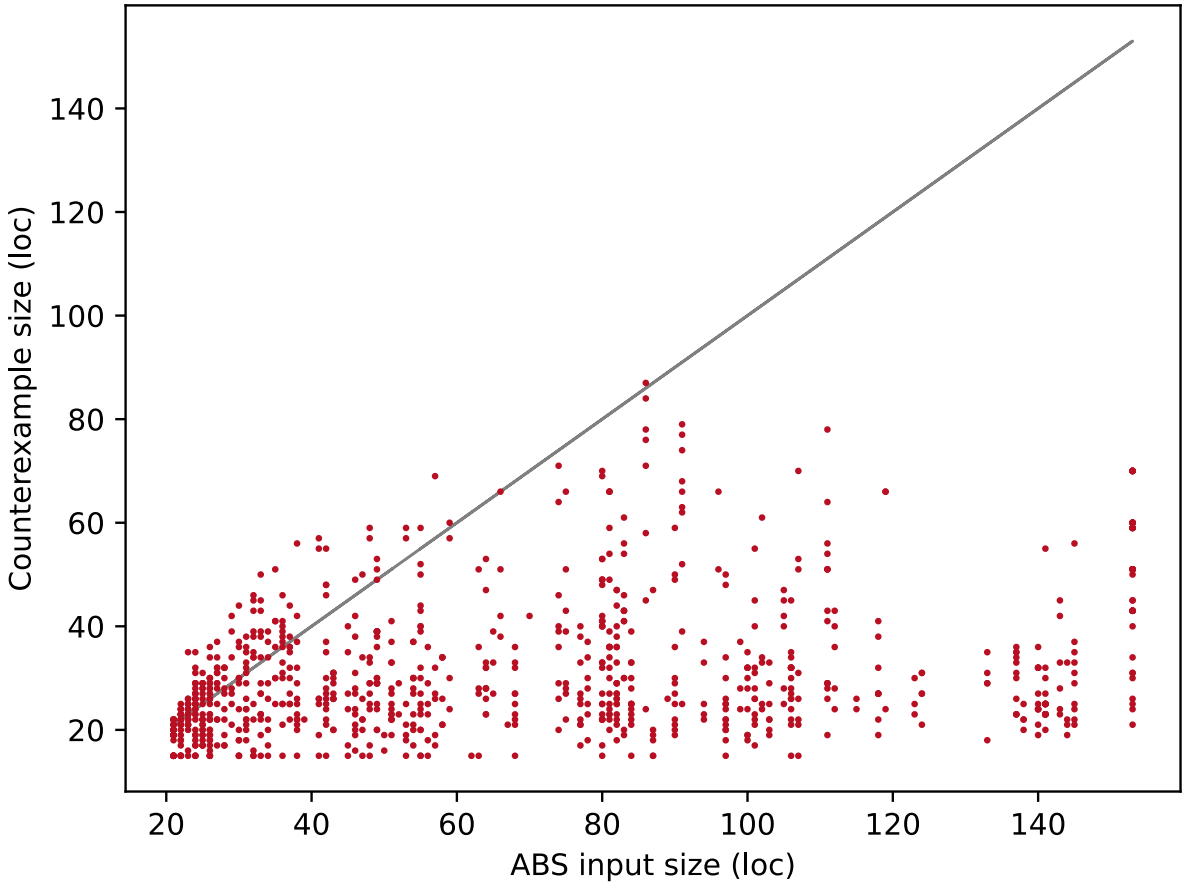

Figure 6.2: Input program size vs. counterexample size in lines of code for 300 randomly generated inputs. The diagonal shows the line of equal input/output size.

generated counterexamples. We also add a 10-minute timeout for verification attempts. The main parameters for program generation that we discussed previously remain unchanged. Counterexample sizes for this dataset are shown in Figure 6.3, which shows the same general trend as the previous dataset. The median counterexample size for this dataset is 26 sloc, which is comparable to the 21 median sloc of counterexamples from the first dataset excluding those from the linearly growing section of the plot below 60 sloc input size.

We feel that these results confirm our method's ability to create counterexamples that are both lower in complexity than the original input and remain within reasonable size limits regardless of input size. As the generated inputs contain only a single method (not counting trivial method stubs included to generate method calls), what we measure in the above experiments is the size reduction of this single method. In a real-world scenario with multiple methods and classes, the complexity reduction achieved by the counterexample would be even larger because the counterexample does not include or require knowledge of any other classes or method implementations.

While the number of counterexamples generated for each input is most likely not representative for real-world verification tasks due to the lack of specification in our generated inputs, we include an analysis of the number

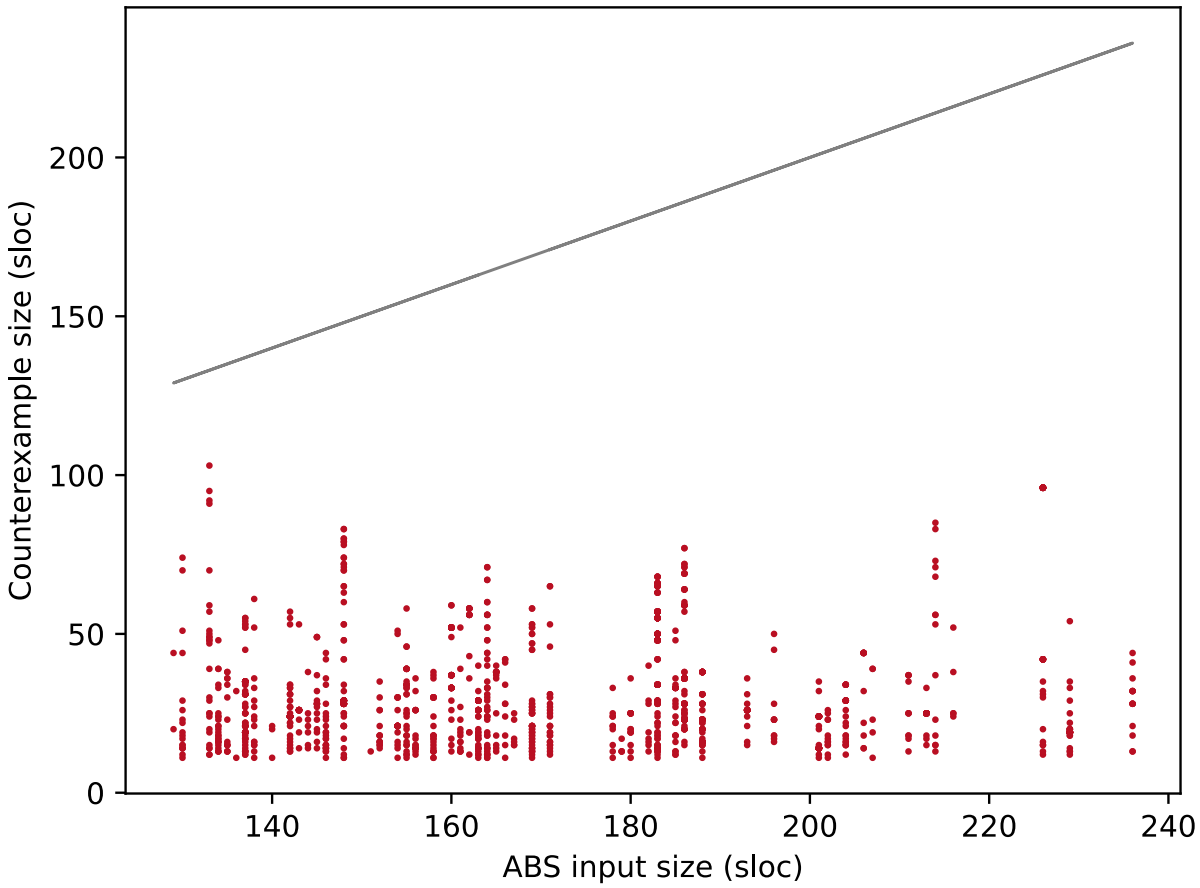

Figure 6.3: Input program size vs. counterexample size in lines of code for 100 randomly generated inputs above 130 loc. The diagonal shows the line of equal input/output size.

of counterexamples relative to the input size in Figure 6.4. This indicates that the 52 counterexamples generated for one of our inputs are an extreme outlier and that the number of counterexamples tends to grow moderately with the size of the input.

The data presented in these plots, as well as all generated ABS programs and corresponding counterexamples are available online at `https://github.com/rec0de/crowbar-investigator-experiments`.

6.2 Clarity and Abstraction

The remaining two goals of *clarity* and *abstraction* are by their very nature much harder to quantify than the previous ones. As such, we will not be able to provide a statistical analysis of these properties — instead, we will attempt to provide an intuition of the level of abstraction our counterexamples provide and how they can help in debugging a verification attempt. In doing so, we will perform a small case study in which we

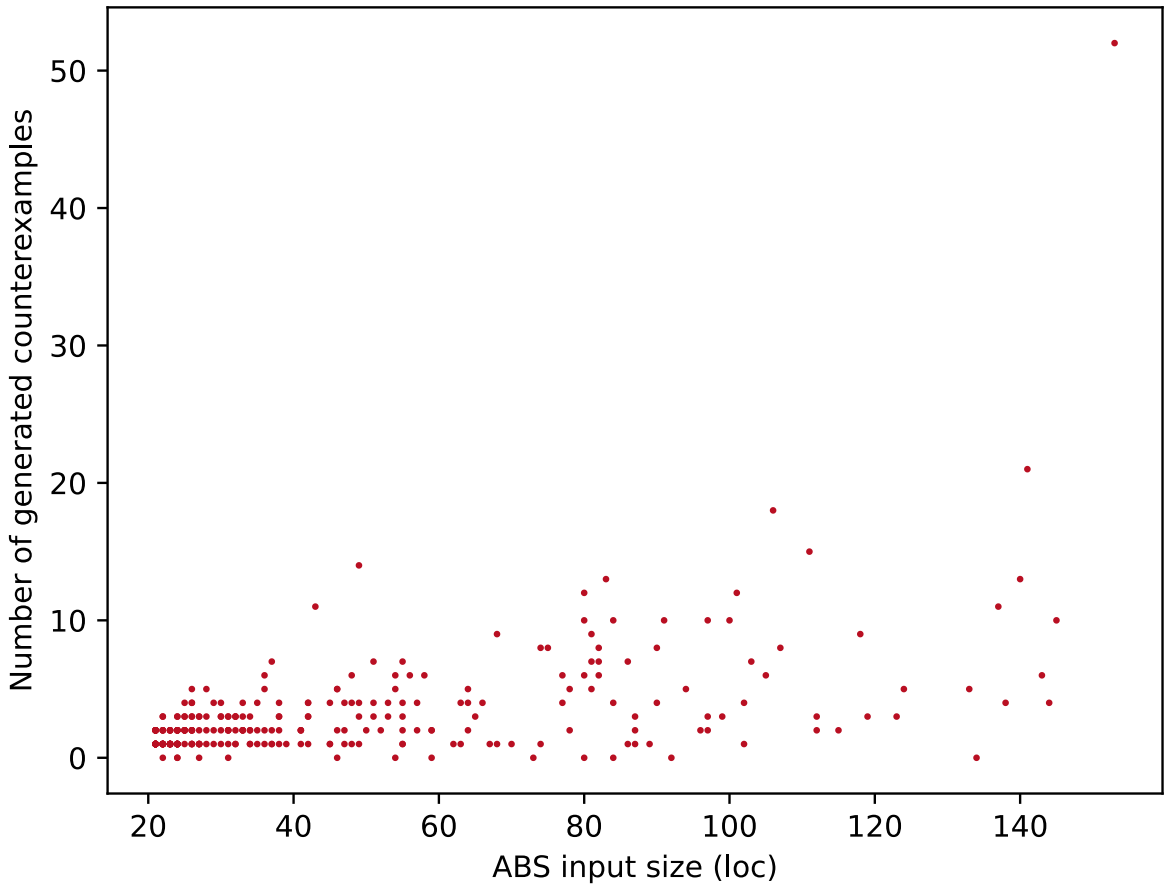

Figure 6.4: Plot of the number of counterexamples generated for each input in relation to input size. Inputs with no counterexamples indicate 'accidental' successful proofs, frequently due to `await False;` statements introducing the false statement as a premiss.

walk through the verification process for an example program based on the temperature sensor example we introduced in section 2.1.

The working example for this section is shown in Listing 6.5. It contains several common types of errors in both implementation and specification.[4] Remembering the iterative nature of the verification process, we will start by attempting verification and fix upcoming issues one by one.

[4]The example in its entirety is also available online at `https://github.com/rec0de/crowbar-investigator-experiments/blob/master/casestudy/faulty-example.abs`. If you would like to try your hand at finding and fixing these issues — with or without the help of counterexamples — now would be a good time to pause and do so.

```
module CaseStudy;

data Spec = ObjInv(Bool) | Ensures(Bool) | Requires(Bool) | WhileInv(Bool);

interface Sensor {
  Int getReading();
}

class Reader {
  Int maxTemp = 0;

  [Spec: Ensures(this.maxTemp >= old(this.maxTemp))]
  Unit read(Sensor s) {
    Fut<Int> reading = s!getReading();
    await reading?;
    Int temp = reading.get;
    if(temp > this.maxTemp) {
      this.maxTemp = temp;
    }
  }

  [Spec: Requires(s != null)]
  [Spec: Ensures(this.maxTemp == old(this.maxTemp))]
  Int readAvg(Sensor s, Int sample) {
    Int i = 0;
    Int sum = 0;

    [Spec: WhileInv(s != null)]
    while(i < sample) {
      i = i + 1;
      Fut<Int> reading = s!getReading();
      await reading?;
      Int value = reading.get;
      sum = sum + value;
    }

    Int avg = this.div(sum, sample);
    return avg;
  }

  [Spec: Requires(b > 0)]
  [Spec: Ensures(result * b <= a && (result + 1) * a > b)]
  [Spec: Ensures(this.maxTemp == old(this.maxTemp))]
  Int div(Int a, Int b) {
    Int div = 0;
    Int remainder = a;

    [Spec: WhileInv(div * b + remainder == a)]
    [Spec: WhileInv(this.maxTemp == old(this.maxTemp))]
    while(remainder >= b) {
      remainder = remainder - b;
      div = div + 1;
    }
    return div;
  }
}
```

Listing 6.5: An ABS program with (faulty) specification.

Our first attempt at full verification of the given program predictably fails and generates six different counterexamples. The first of these highlights a very simple issue and is correspondingly short: As shown in Listing 6.6[5], the proof fails right at the beginning of the *read* method because the sensor object *s* on which we call the *getReading* method could not be shown to be non-null. Recognizing this issue, we can simply add a *requires* clause to the *read* method stating that *s* must be non-null.

```
1 // Snippet from: read
2 // Assume the following pre-state:
3 String s = null;
4 // End of setup
5
6 // Proof failed here. Trying to show:
7 // Null-check: !(s=0)
8 // Failed to show the following sub-obligations:
9 // !(s=0)
```

Listing 6.6: Relevant parts of the first counterexample generated for the program in 6.5

The second counterexample turns out to be a bit more complex than the first, although we have already seen its underlying issue in a previous section. In line 7 of Listing 6.7, we clearly see that while the method is waiting for the future *reading* to resolve, the *maxTemp* field could be modified by another method. We could, then, read a temperature that is below the previous maximum and incorrectly write it to the *maxTemp* field in lines 11–13. Remember that this bug was rather non-obvious when we discussed it in the introduction of symbolic execution (section 2.2) — the counterexample made all the subtle concurrency assumptions explicit in a clear example of how the method might be executed.
To fix this issue, we will introduce a local backup of the *maxTemp* field (see lines 12–19 in Listing 6.14). This backup, which cannot be modified by other processes, is used within the method and written back to the field before returning. Note that this introduces a previously impossible race-condition causing lost updates which, however, do not violate our specification.
The third counterexample pertains to the same bug so we will not discuss it in detail — the only difference is that no false new maximum temperature is measured and written to the field. Instead, the modified value from line 7 remains.

With the next counterexample, we move on to the *readAvg* method, which is supposed to average a given number of readings into a single measurement. As crowbar does not yet support floating-point numbers, we use a custom integer division method to compute the average. As we have no way of specifying what the correct average of the readings would be, we restrict ourselves to only showing noninterference with the *maxTemp* field we have seen before.
The first issue we find, however, is related to the division method call. As stated in line 14 of Listing 6.8, the proof fails to show the method precondition of *div*. In fact, the assignment in line 3 sets *sample* to zero, which would result in a division by zero in the call to *div*. With a glance at our specification, we see that we never specified the *sample* parameter to be non-zero, so we add $sample > 0$ to the method preconditions and the loop invariant and move on.

[5] All of the following counterexamples are presented as generated, with some whitespace adjustments and parts of the frame removed for better readability.

```
// Snippet from: read
// Fut reading = s!getReading();
String reading = "fut_1";

// await reading?;
// Assume the following assignments while blocked:
this.maxTemp = -2;
// End assignments

// Int temp = (reading).get;
Int temp = -1;
if((temp > this.maxTemp)){
  this.maxTemp = temp;
}
// return unit
// Proof failed here. Trying to show:
// Method postcondition: (heap.maxTemp>=old.maxTemp)
// Object invariant: true
// Failed to show the following sub-obligations:
// (select(store(heap, this.maxTemp, valueOf(fut_1)), this.maxTemp)>=heap.maxTemp)
```

Listing 6.7: Relevant parts of the second counterexample generated for the program in 6.5

```
// Snippet from: readAvg
// Assume the following pre-state:
Int sample = 0;
Int i = 0;
String s = "object_?(1)";
// End of setup

while((i < sample)){}
// Known true:
// Negated loop guard: !((i < sample))
// Loop invariant: !(s=0)

// Proof failed here. Trying to show:
// Method precondition: (sample>0)
// Failed to show the following sub-obligations:
// (sample>0)
```

Listing 6.8: Relevant parts of the fourth counterexample generated for the program in 6.5

Perhaps more interesting than the last example, the fifth counterexample shows an actual problem with the non-interference property we specified. When looking at the counterexample in Listing 6.9, one thing that should stand out is the unusual rendering of the *maxTemp* field in line 2. This is due to an interesting edge case: The code executed as part of this counterexample contains no reference to *maxTemp*, and the specification carries no information about its type. Therefore, the program reconstruction logic does not have access to the true type of the field and falls back to rendering it as a string, including the underlying integer value.
This quirk does not change anything about how we read the counterexample, though: We can see that *maxTemp* has some given value in the pre-state as shown in line 2. At the end of the method, *maxTemp* has a different value, namely `unknownType(0)`, thereby violating our specification (see line 29). Note also that the assignment in line 21 does not change the value of *maxTemp*, as we have specified for the *div* method. Looking at the beginning of the counterexample, we can find the reason for this behaviour: We are in a state after the while-loop was executed, and the only things we know to be true are the two statements in lines 8 and 9. In other words, the code executed in the loop body might have changed *maxTemp* to an unknown value. To fix this, we strengthen the loop invariant to include `this.maxTemp == old(this.maxTemp)`.

```
1  class CeFrame implements Ce {
2      String maxTemp = "unknownType(1)";
3
4      Unit ce() {
5          // Snippet from: readAvg
6          // while((i < sample)){}
7          // Known true:
8          // Negated loop guard: !((i < sample))
9          // Loop invariant: !(s=0)
10
11         // Assume the following pre-state:
12         Int sum = 0;
13         Int sample = -1;
14         Int i = -1;
15         String s = "object_?(1)";
16         this.maxTemp = "unknownType(0)";
17         // End of setup
18
19         // Int avg = this.div(sum, sample);
20         // Assume the following assignments while blocked:
21         this.maxTemp = "unknownType(0)";
22         // End assignments
23
24         Int avg = 0;
25         println(toString(avg)); // return statement
26         // Evaluates to: 0
27
28         // Proof failed here. Trying to show:
29         // Method postcondition: (heap.maxTemp=old.maxTemp)
30         // Object invariant: true
31         // Failed to show the following sub-obligations:
32         // (select(anon(heap), this.maxTemp)=old.maxTemp)
33     }
34 }
```

Listing 6.9: Relevant parts of the fifth counterexample generated for the program in 6.5

The last counterexample for this proof attempt is for the *div* method. From the last lines of output in Listing 6.10, we can tell that the proof failed trying to show the properties we expect for the quotient of a and b. This is odd, because we know the loop invariant to hold (line 5), stating that $div \cdot b + remainder = a$. With the remainder being smaller than b, this should mean that div is the divisor of a and b. With the values given in the counterexample, we can see that $0 \cdot 0 + (-1) = -1$ does indeed hold. The problem is that the remainder should never be negative, which we forgot to specify in the loop invariant.

```
1  // Snippet from: div
2  // while((remainder >= b)){}
3  // Known true:
4  // Negated loop guard: !((remainder >= b))
5  // Loop invariant: ((((div*b)+remainder)=a)) /\ ((heap.maxTemp=old.maxTemp))
6
7  // Assume the following pre-state:
8  Int remainder = -1;
9  Int b = 0;
10 Int div = 0;
11 Int a = -1;
12 // End of setup
13
14 println(toString(div)); // return statement
15 // Evaluates to: 0
16 // Proof failed here. Trying to show:
17 // Method postcondition: ((((result*b)<=a)) /\ ((((result+1)*a)>b))) /\ ((heap.maxTemp=old.
       maxTemp))
18 // Object invariant: true
19 // Failed to show the following sub-obligations:
20 // ((div*b)<=a)
21 // (((div+1)*a)>b)
```

Listing 6.10: Relevant parts of the sixth counterexample generated for the program in 6.5

With that, we have now addressed all six counterexamples and are ready to attempt verification anew. We do not include the full program with the changes we applied here, but you can find it online at `https://github.com/rec0de/crowbar-investigator-experiments/blob/master/casestudy/first-pass-res.abs`. Our second verification attempt produces only three new counterexamples, which is a sure sign of progress.

The first of these new counterexamples — shown in Listing 6.11 — is related to the strengthened loop invariant in *readAvg*. Despite us specifying that the statements in the loop body should never modify *maxTemp*, the counterexample generator has found a scenario in which they do just that, violating the loop invariant preservation. Specifically, we are once again reminded of the concurrency model of ABS in line 19, where the field is modified while we release control of the processing unit when waiting for *reading* to resolve.
Unfortunately, there is no elegant way to fix this problem as crowbar, at its current stage, does not allow us to express that no method that could *possibly* be called while we release control modifies the *maxTemp* field. However, we assume in our current setup that the temperature sensors are remote objects in a different *cog*. Remembering the semantics of the *get* expression [1, sec. 4.2], we can just remove the *await* statement in this case. When the *get* expression in line 22 is reached before the value of *reading* is available, the process will block *without releasing control of the processor*. This means no other process in the same cog can modify *maxTemp* and execution can continue as soon as the value of *reading* becomes available from the remote cog.

The remaining two counterexamples both concern the *div* method. After previously restricting the *remainder*

```
1  // Snippet from: readAvg
2  // Assume the following pre-state:
3  Int sample = 1;
4  Int i = 0;
5  String s = "object_?(1)";
6  // End of setup
7
8  // Known true:
9  // Loop guard: (i < sample)
10 // Loop invariant: ((!(s=0)) /\ ((sample>0))) /\ ((heap.maxTemp=old.maxTemp))
11 // while((i < sample)) {
12 {
13     i = (i + 1);
14     // Fut reading = s!getReading();
15     String reading = "fut_4";
16
17     // await reading?;
18     // Assume the following assignments while blocked:
19     this.maxTemp = -1;
20     // End assignments
21
22     // Int value = (reading).get;
23     // Future value irrelevant or unavailable, using default:
24     Int value = 0;
25     Int sum = (sum + value);
26 }
27 // Proof failed here. Trying to show:
28 // Loop invariant: ((!(s=0)) /\ ((sample>0))) /\ ((heap.maxTemp=old.maxTemp))
29 // Failed to show the following sub-obligations:
30 // (select(anon(heap), this.maxTemp)=old.maxTemp)
```

Listing 6.11: Relevant parts of the seventh counterexample generated for the program in 6.5

variable to non-negative values in the loop invariant, the proof now fails to show that the new loop invariant holds initially. From the proof obligation analysis starting at line 10 of Listing 6.12, we can tell both that the invariant could not be shown as well as that the part that causes issues is the one concerning the remainder — specifically with the remainder evaluated to *a*. The statements leading up to the loop in line 9 confirm this: With *a* set to a negative value, the remainder variable is initialized to that negative number, violating our assumption.
At this point, we could re-write our division code to properly handle negative inputs. This being an example, however, we will simply assume that we measure temperatures in degrees Kelvin and thus never have to deal with negative numbers. With this assumption, we can specify the *getReading* method to always return non-negative values and require the *a* parameter of *div* to be non-negative as well. As we divide the sum of several readings, we will also have to propagate the information about the readings to the *sum* variable by adding a clause specifying $sum >= 0$ to the loop invariant in *readAvg*.[6] All of these modifications can be seen in the fully corrected program in Listing 6.14.

The next counterexample in Listing 6.13 shows an issue with the postcondition of the method. Looking at the state information in lines 9–12, we can see that the generator chose $a = 0$ and $b = 1$. This matches the return value shown in line 16 as $0/1 = 0$. Still, the proof failed to show part of the postcondition we specified

[6] These adjustments could, of course, also be discovered in a more iterative fashion using several more verification attempts and counterexamples. We chose to omit these steps for brevity here.

```
// Snippet from: div
// Assume the following pre-state:
Int a = -1;
Int b = 1;
// End of setup

Int div = 0;
Int remainder = a;
while((remainder >= b)) { }
// Proof failed here. Trying to show:
// Loop invariant: (((((div*b)+remainder)=a)) /\ ((remainder>=0))) /\ ((heap.maxTemp=old.
    maxTemp))
// Failed to show the following sub-obligations:
// (a>=0)
```

Listing 6.12: Relevant parts of the eigth counterexample generated for the program in 6.5

— $(div + 1) \cdot a > b$, to be precise. When comparing this part of the postcondition to the part that did not cause any problems, it becomes fairly obvious that we mistakenly swapped a and b in the second part of the postcondition which should read $(div + 1) \cdot b > a$.[7]

```
// Snippet from: div
// while((remainder >= b)){}
// Known true:
// Negated loop guard: !((remainder >= b))
// Loop invariant: (((((div*b)+remainder)=a)) /\ ((remainder>=0))) /\ ((heap.maxTemp=old.
    maxTemp))

// Assume the following pre-state:
Int remainder = 0;
Int b = 1;
Int div = 0;
Int a = 0;
// End of setup

println(toString(div)); // return statement
// Evaluates to: 0
// Proof failed here. Trying to show:
// Method postcondition: ((((result*b)<=a)) /\ ((((result+1)*a)>b))) /\ ((heap.maxTemp=old.
    maxTemp))
// Object invariant: true
// Failed to show the following sub-obligations:
// (((div+1)*a)>b)
```

Listing 6.13: Relevant parts of the ninth counterexample generated for the program in 6.5

Once again, we have worked through all counterexamples and are now ready to start a new verification attempt. With the new modifications to the code and its specification, this third attempt succeeds and we have verified the program. The final version with all the discussed adjustments is shown in Listing 6.14.

Across all of these counterexamples, we have needed very little knowledge of the calculus behind crowbar or the finer details of the ABS concurrency model — with the help of counterexamples, we do not need to know

[7] While this might seem like a contrived example, I actually made this mistake when developing the case study.

which invariants must hold at which points or where possible state modifications might occur. Instead, all of these hidden details are made explicit in the counterexample annotations. Even in the case of typically hard to spot errors in the specification like the swapped variables from Listing 6.13, the counterexample narrows down the search space to a single subexpression and provides us with concrete values that show that we indeed made a mistake in writing this part of the specification. Considering these aspects and the intuition gained from walking through the verification process for an entire program, we feel that our counterexample generation method provides a sufficient level of abstraction such that non-expert programmers can verify their ABS code with minimal knowledge of the verification system's internals.

As for clarity, we observe that each presented counterexample highlights a single, discrete issue that can usually be fixed or at least understood in isolation. Thanks to the annotations showing the problematic proof obligations as well as their type, this issue is fairly obvious at least in the examples we have seen. The list of problematic sub-obligations in particular has shown to be very effective at reducing the search-space for debugging, for example in Listings 6.12 and 6.13.
We do concede, however, that some counterexamples could be made to be more concise. The counterexample in Listing 6.11, for example, includes assignments to the loop index and sensor object that are not relevant to the proof and in Listing 6.9 the rendering of the synchronous call to *div* includes an assignment that does not actually change the state. Removing such statements or rendering them as comments only could help reduce visual clutter and further improve counterexample clarity. We will leave this as an area for future work.
That being said, it is hard to overstate the improvement in clarity over the default verification output, which boils down to boolean return values, or a raw dump of the symbolic execution tree. For the sake of illustration, we include an excerpt from a symbolic execution tree as produced by crowbar for the initial version of our program in Listing 6.15.

```
module CaseStudy;
data Spec = ObjInv(Bool) | Ensures(Bool) | Requires(Bool) | WhileInv(Bool);

interface Sensor { [Spec: Ensures(result >= 0)] Int getReading(); }

class Reader {
  Int maxTemp = 0;

  [Spec: Requires(s != null)]
  [Spec: Ensures(this.maxTemp >= old(this.maxTemp))]
  Unit read(Sensor s) {
    Int localMax = this.maxTemp;
    Fut<Int> reading = s!getReading();
    await reading?;
    Int temp = reading.get;
    if(temp > localMax) {
      localMax = temp;
    }
    this.maxTemp = localMax;
  }

  [Spec: Requires(s != null && sample > 0)]
  [Spec: Ensures(this.maxTemp == old(this.maxTemp))]
  Int readAvg(Sensor s, Int sample) {
    Int i = 0;
    Int sum = 0;

    [Spec: WhileInv(s != null && sample > 0 && sum >= 0 && this.maxTemp == old(this.maxTemp))]
    while(i < sample) {
      i = i + 1;
      Fut<Int> reading = s!getReading();
      Int value = reading.get;
      sum = sum + value;
    }
    Int avg = this.div(sum, sample);
    return avg;
  }

  [Spec: Requires(b > 0 && a >= 0)]
  [Spec: Ensures(result * b <= a && (result + 1) * b > a)]
  [Spec: Ensures(this.maxTemp == old(this.maxTemp))]
  Int div(Int a, Int b) {
    Int div = 0;
    Int remainder = a;

    [Spec: WhileInv(div * b + remainder == a && remainder >= 0)]
    [Spec: WhileInv(this.maxTemp == old(this.maxTemp))]
    while(remainder >= b) {
      remainder = remainder - b;
      div = div + 1;
    }
    return div;
  }
}
```

Listing 6.14: A corrected version of our example program that passes verification

```
1 b:Int>0
2 ==>
3 {last:Heap := heap:Heap}{old:Heap := heap:Heap}[div:Int = 0;remainder:Int = a:Int;while{pp:7}(
      >=(remainder:Int,b:Int) ){ remainder:Int = -(remainder:Int,b:Int);div:Int = +(div:Int,1)
    };return div:Int || ((result:<UNKNOWN>*b:Int<=a:Int) /\ (result:<UNKNOWN>+1*a:Int>b:Int))
    /\ (select(heap:Heap,this.maxTemp_f : <UNKNOWN>)=select(old:Heap,this.maxTemp_f : <UNKNOWN
    >)), true]
4   b:Int>0
5 ==>
6 {last:Heap := heap:Heap}{old:Heap := heap:Heap}{div:Int := 0}[remainder:Int = a:Int;while{pp
    :7}( >=(remainder:Int,b:Int) ){ remainder:Int = -(remainder:Int,b:Int);div:Int = +(div:Int
    ,1) };return div:Int || ((result:<UNKNOWN>*b:Int<=a:Int) /\ (result:<UNKNOWN>+1*a:Int>b:Int
    )) /\ (select(heap:Heap,this.maxTemp_f : <UNKNOWN>)=select(old:Heap,this.maxTemp_f : <
    UNKNOWN>)), true]
7     b:Int>0
8 ==>
9 {last:Heap := heap:Heap}{old:Heap := heap:Heap}{div:Int := 0}{remainder:Int := a:Int}[while{pp
    :7}( >=(remainder:Int,b:Int) ){ remainder:Int = -(remainder:Int,b:Int);div:Int = +(div:Int
    ,1) };return div:Int || ((result:<UNKNOWN>*b:Int<=a:Int) /\ (result:<UNKNOWN>+1*a:Int>b:Int
    )) /\ (select(heap:Heap,this.maxTemp_f : <UNKNOWN>)=select(old:Heap,this.maxTemp_f : <
    UNKNOWN>)), true]
```

Listing 6.15: Excerpt from a symbolic execution tree produced by crowbar in verbose mode

7 Conclusion

In conclusion, we defined the four goals of *minimal size*, *minimal dependency*, *high clarity* and *high abstraction* for counterexample generation methods built to support formal verification tasks. We outlined a method for generating such counterexamples consisting of the four stages *program reconstruction*, *model integration*, *context removal* and *counterexample generation*. This method is abstract and can be applied to any formal verification system that uses symbolic execution in conjunction with an off-the-shelf SMT solver. We then described in detail our implementation of said method for formal verification of ABS with crowbar and the design considerations that arise from the characteristics of the language. The process of adapting our general method to another language can be expected to follow a similar pattern.
We continued to extend our counterexample method with human-readable annotations, aiming to improve the readability and clarity of the generated counterexamples. While this, conceptually, is also applicable to other languages or verification systems, the nature and placement of the annotations are specific to ABS and crowbar.
Finally, we evaluated the performance of our implementation with regards to the previously defined quality goals. In doing so, we generated counterexamples for random ABS programs, finding that the counterexample size tends to grow linearly with the input size for small inputs, but stays effectively constant for large inputs. We also found that almost all counterexamples have lower complexity than their corresponding inputs.
Investigating the aspects of clarity and abstraction, we walked through the iterative verification process using an exemplary ABS program and fixed any arising problems using the counterexamples generated by our method. We concluded that our method provides sufficient levels of clarity and abstraction to be a valuable tool for verification debugging even or especially for non-experts.

7.1 Counterexample-Driven Verification

As we have established in the introduction to this work, formal verification is a highly iterative process. Without proper tool support, however, the iterations that make up this process are poorly defined and require guesswork and labour-intensive investigation by the user. We hope that the counterexample generation method we outlined can help enable a verification workflow with tighter and more well-defined iteration loops. Starting from an initial implementation and method-level specification, counterexamples can be used to incrementally add further specification and discover bugs or edge-cases, much like we have done in the case study in section 6.2.

This process is in large parts analogous to *test-driven development* (TDD) [15], in which an implementation is developed in short iterations based on previously designed testcases or *design by contract* [9], in which desired behaviour is specified using executable assertions in the code.

What these processes have in common is that they use some form of external specification, be it in the form of formal specification or vectors of test data, to iterate towards a working implementation. This specification is

usually written before large parts of the program are. Where the methods differ is in how the programmer is guided through these iterations. For example, using crowbar without the crowbar investigator to check the validity of a program's specification could be seen as a basic form of design by contract — but it does not do much except notify the user that an assertion failed. Failing testcases in TDD, on the other hand, provide the user with concrete input vectors that caused undesired behaviour. The counterexamples we generate in our method go beyond that and show a complete, annotated version of the relevant execution path alongside concrete input values, providing the user with significantly more information to identify and fix the issue at hand.

While we could use our counterexample generator to perform true *counterexample-driven development* in the style of TDD or design by contract, starting with a minimal base implementation and iterating until the verification succeeds, such a workflow would be quite cumbersome. Counterexamples generated for empty method stubs, for example, are rather dull and do little to guide the programmer towards a working implementation. Instead, we imagine our counterexample generator to be used alongside a traditional development workflow such as TDD in what could be described as *counterexample-driven verification*: While the traditional process is used to develop a base implementation, the counterexample-driven process is mainly used to guide the user through formal verification, thus combining the strengths of both methods. TDD and counterexample-driven verification in particular make for a very natural union here because of their similar structures.

In this hybrid process, test-driven development can be seen as the coarse-grained tool which is used to quickly iterate towards a working implementation that somewhat matches our intuitive understanding of the task. The counterexample-driven part, then, is a more fine-grained tool used to eliminate edge-cases and iterate more slowly towards an implementation that fully conforms to the specification and can be verified.

7.2 Future Work

There are many opportunities for further research into similar and improved counterexample generation methods. First and foremost, it would be interesting to see how a similar method could be applied to more complex formal verification frameworks such as the KeY System [2]. The much more complex nature of these systems brings with it a set of new challenges, among them the question if off-the-shelf SMT solvers can produce useful models for proofs heavily relying on first-order logic. Adapting our method to a new environment also requires finding suitable representations for rules and proof obligations unique to the target verification method. For systems like KeY, this means finding ways to represent more complex specifications, such as *measuredBy*, *accessible* and *assignable* clauses [2, chapter 9] as well as supporting the much larger set of symbolic execution rules.
Despite these challenges, there should be no fundamental barriers to a similar method being implemented for more complex verification systems.

As the crowbar tool is only in the early stages of development at this time, there is also work to be done in adapting our method to upcoming crowbar features. Support for algebraic data types especially would require additions to the context removal method to generate appropriate replacements for these types. Other possible changes requiring modifications to our method include support for exceptions and rules for more ABS statements (e.g. the *suspend* statement [1, sec. 6.7]).

Our method, as outlined in section 3, requires tight integration with the symbolic execution engine of the verification system to collect and store information relevant for the counterexample generation. This comes

with the obvious drawback that implementing counterexample generation requires significant changes to the symbolic execution engine. It would therefore be very valuable to develop a similar method that operates only on symbolic execution trees as produced by an unmodified execution engine. This would reduce the coupling between the two components and enable more generic and interchangeable counterexample generators. Required information absent from the execution tree could instead be sourced directly from the program source code.

Lastly, we have seen in the size analysis of counterexamples in section 6.1.2 that there is still room for improvement with regards to minimal counterexample size. We have also noted previously that redundant or irrelevant statements can form visual clutter and reduce clarity. Future iterations of counterexample generation methods might benefit from counterexample / testcase minimization techniques, for example by detecting and removing statements or memory locations not relevant to the current proof. We have already seen a problem arising from missing information in the reconstruction of some statements not relevant to the current proof obligation (see section 4.2). It should be possible to use this behaviour explicitly to detect and remove redundant statements using the existing infrastructure of the crowbar investigator. Alternatively, methods previously described in the context of unit tests [23, 19] or other formal verification systems such as Promela/SPIN [11] could also be worth exploring.

Bibliography

[1] ABS Development Team. *ABS Documentation*. https://abs-models.org/manual. [Online; accessed 5-September-2020]. 2020.

[2] Wolfgang Ahrendt et al., eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, Dec. 16, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6. URL: http://dx.doi.org/10.1007/978-3-319-49812-6. published.

[3] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

[4] Clark Barrett et al. "CVC4". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177. ISBN: 978-3-642-22110-1.

[5] Bernhard Beckert, Sarah Grebing, and Mattias Ulbrich. "An Interaction Concept for Program Verification Systems with Explicit Proof Object". In: *Hardware and Software: Verification and Testing*. Ed. by Ofer Strichman and Rachel Tzoref-Brill. Cham: Springer International Publishing, 2017, pp. 163–178. ISBN: 978-3-319-70389-3.

[6] David R. Cok. "jSMTLIB: Tutorial, Validation and Adapter Tools for SMT-LIBv2". In: *NASA Formal Methods*. Ed. by Mihaela Bobaru et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 480–486. ISBN: 978-3-642-20398-5.

[7] Quoc Huy Do, Richard Bubel, and Reiner Hähnle. "Exploit Generation for Information Flow Leaks in Object-Oriented Programs". In: *ICT Systems Security and Privacy Protection*. Ed. by Hannes Federrath and Dieter Gollmann. Cham: Springer International Publishing, 2015, pp. 401–415. ISBN: 978-3-319-18467-8.

[8] Christian Engel and Reiner Hähnle. "Generating Unit Tests from Formal Proofs". In: *Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland*. Ed. by Yuri Gurevich and Bertrand Meyer. Vol. 4454. LNCS. Springer, Jan. 1, 2007. published.

[9] Yishai A. Feldman. "Extreme Design by Contract". In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Michele Marchesi and Giancarlo Succi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 261–270. ISBN: 978-3-540-44870-9.

[10] Erich Gamma et al. *Design Patterns Elements of reusable object-oriented sofware*. Addison-Wesley, 1994. ISBN: 0201633612.

[11] Paul Gastin, Pierre Moro, and Marc Zeitoun. "Minimization of Counterexamples in SPIN". In: *Model Checking Software*. Ed. by Susanne Graf and Laurent Mounier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 92–108. ISBN: 978-3-540-24732-6.

[12] Martin Hentschel, Reiner Hähnle, and Richard Bubel. “The Interactive Verification Debugger: Effective Understanding of Interactive Proof Attempts”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 846–851. ISBN: 9781450338455. DOI: 10.1145/2970276.2970292. URL: https://doi.org/10.1145/2970276.2970292.

[13] Mihai Herda et al. “Understanding Counterexamples for Relational Properties with DIbugger”. In: *Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6-7th April 2019*. Ed. by Emanuele De Angelis et al. Vol. 296. EPTCS. 2019, pp. 6–13. DOI: 10.4204/EPTCS.296.4. URL: https://doi.org/10.4204/EPTCS.296.4.

[14] Mihai Herda et al. “Verification-Based Test Case Generation for Information-Flow Properties”. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC ’19. Limassol, Cyprus: Association for Computing Machinery, 2019, pp. 2231–2238. ISBN: 9781450359337. DOI: 10.1145/3297280.3297500. URL: https://doi.org/10.1145/3297280.3297500.

[15] David Janzen and Hossein Saiedian. “Test-driven development concepts, taxonomy, and future direction”. In: *Computer* 38.9 (2005), pp. 43–50.

[16] Einar Broch Johnsen et al. “ABS: A Core Language for Abstract Behavioral Specification”. In: *Formal Methods for Components and Objects*. Ed. by Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 142–164. ISBN: 978-3-642-25271-6.

[17] Eduard Kamburjan. “Behavioral Program Logic”. In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Ed. by Serenella Cerrito and Andrei Popescu. Cham: Springer International Publishing, 2019, pp. 391–408. ISBN: 978-3-030-29026-9.

[18] Eduard Kamburjan, Marco Scaletta, and Nils Rollshausen. *Crowbar*. https://github.com/Edkamb/crowbar-tool. [Online; accessed 5-September-2020]. 2020.

[19] Andreas Leitner et al. “Efficient Unit Test Case Minimization”. In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 417–420. ISBN: 9781595938824. DOI: 10.1145/1321631.1321698. URL: https://doi.org/10.1145/1321631.1321698.

[20] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[21] Fadia Nemer et al. “PapaBench: a Free Real-Time Benchmark”. In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET’06)*. Ed. by Frank Mueller. Vol. 4. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. ISBN: 978-3-939897-03-3. DOI: 10.4230/OASIcs.WCET.2006.678. URL: http://drops.dagstuhl.de/opus/volltexte/2006/678.

[22] Nils Rollshausen. *MockMAVL*. https://github.com/rec0de/mockMAVL. [Online; accessed 25-September-2020]. 2020.

[23] A. Zeller and R. Hildebrandt. “Simplifying and isolating failure-inducing input”. In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200.