Improving Counterexample Quality from Failed Program Verification

Li Huang

Chair of Software Engineering Schaffhausen Institute of Technology Schaffhausen, Switzerland li.huang@sit.org Bertrand Meyer Chair of Software Engineering Schaffhausen Institute of Technology Schaffhausen, Switzerland bm@sit.org Manuel Oriol Chair of Quantum Software Engineering Schaffhausen Institute of Technology Schaffhausen, Switzerland mo@sit.org

Abstract—In software verification, a successful automated program proof is the ultimate triumph. The road to such success is, however, paved with many failed proof attempts. The message produced by the prover when a proof fails is often obscure, making it very hard to know how to proceed further. The work reported here attempts to help in such cases by providing immediately understandable counterexamples. To this end, it introduces an approach called Counterexample Extraction and Minimization (CEAM). When a proof fails, CEAM turns the counterexample model generated by the prover into a a clearly understandable version; it can in addition simplify the counterexamples further by minimizing the integer values they contain. We have implemented the CEAM approach as an extension to the AutoProof verifier and demonstrate its application to a collection of examples.

Index Terms—Program Verification, Counterexample, Auto-Proof, Boogie, SMT

I. INTRODUCTION

Deductive program verification performs a rigorous analysis of the correctness of programs with respect to their functional behavior, usually specified formally by contracts (such as pre- and postconditions, can class and loop invariants). The approached has progressed in recent years thanks to the development of powerful proof engines. In practice, however, verifying industrial applications remains difficult. One of the obstacles is the lack of intuitive feedback to understand the reasons why a verification attempt failed. Although in many cases the underlying prover can provide a counterexample containing some usable diagnostic information for debugging, such a counterexample usually contains hundreds of difficultto-interpret lines. Another obstacle to usability is that integer values generated by the prover for the counterexamples are often very large, and hence do not provide programmers with an easy intuitive understanding of what is wrong.

This article presents the Counterexample Extraction and Minimization (CEAM) approach for improving the quality of counterexamples produced when a proof fails, and making them usable for identifying and correcting the underlying bugs. We have implemented CEAM as an extension of the AutoProof environment [1], [23], a static verification platform for contract-equipped Eiffel [15] programs based on Hoarestyle proofs. AutoProof relies on the Boogie proof system [2], [11] and takes advantage of Boogie's underlying SMT (Satisfiability Modulo Theories) solver, by default Z3 [7].

When a proof fails, CEAM exploits the counterexample models (hereinafter referred to simply as models) generated by the SMT solver and generates simple counterexamples in a format more intuitive to programmers. CEAM also provides a minimization mechanism allowing programmers to get simplified counterexamples with integer variables reduced to their minimal possible values. The current version of CEAM supports primitive types (integer, boolean), user-defined types (classes) as well as some commonly used container types (such as arrays and sequences).

Section II illustrates an example of using the CEAM approach. Section III introduces the technologies used in our verification framework. Section IV describes the details of the implementation of the CEAM. We evaluate the applicability of the CEAM through a series of examples in Section V. After a review of related work in Section VI, Section VII concludes the paper with our ongoing work.

II. AN EXAMPLE SESSION

Before exploring the principles and technologies of the CEAM approach, we look at its practical use on a representative example (Fig. 1). The intent of the max function in class MAX is to return into **Result** the maximum element of an integer array a of size a.count. The two postconditions in lines 22 and 23 (labeled by *is_max* and *in_array*) specify this intent: every element of the array should be less than or equal to **Result**; at least one element should be equal to **Result**.

When we try to verify the max function using AutoProof, verification fails and AutoProof returns an error message "Postcondition *is_max* may be violated" (the first row in Fig. 2). Such a generic message tells us that the prover cannot establish the postcondition, but does not enable us to find out why. In this case, programmers can look at the model generated by the Z3 solver to understand the cause of the failure. Deciphering the model is a cumbersome task: the model spans hundreds of lines and is expressed in a cryptic formalism.

In contrast, AutoProof extended with CEAM automatically generates a much simpler counterexample from the model. As displayed in the second row of Fig. 2, the counterexample contains the initial values (on entry of max) of the array's size and of some of its elements. Seeing these concrete values, rather than just the prover's general failure message, helps the programmer conjecture possible reasons for the failure. The values in the counterexample are large, however, too large to give the programmer a direct intuition of the problem at a human scale.

1 class MAX feature

```
2
    max (a: ARRAY [INTEGER]): INTEGER
 3
       require a.count > 0
 4
       local i: INTEGER
 5
       do
 6
            from
 7
                 Result := a [1]; i := 2
 8
            invariant
                 2 \leq i and i \leq a.count + 1
 9
10
                 ∀ j: 1 |..| (i - 1) | a.sequence [j] <= Result
11
                 \exists j: 1 |..| (i - 1) | a.sequence [j] = Result
12
            unt i 1
13
                 i \ge a.count
14
            loop
15
                 if a [i] > Result then
16
                      Result := a [i]
17
                 end
18
                 i := i + 1
19
            variant a.count - i
20
            end
21
        ensure
22
            is_max: ∀ j: 1 |..| a.count | a.sequence [j] <= Result
23
            in_array: ∃ j: 1 |..| a.count | a.sequence [j] = Result
24
        end
```

25 end

Fig. 1. MAX is a class that finds the maximum element of an integer array; a fault (the exit condition at line 13 is incorrect) is injected to the code for presentation purposes.

To provide a more intuitive illustration, CEAM allows the programmer to query AutoProof further to obtain a minimal counterexample in the last row of Fig. 2, where integer variables have been reduced to their minimal possible values.

AutoProof 8								
۲	Verify - 🔳 🛃 2 Su	ccessful 🚡 1 F	ailed 🛕 0 Errors	М.				
	Class	Feature	Information					
-6	MAX	max	Postcondition is maxmay be violated.					
3			Counterexample: a.count = 11800, a [1] = 0, a [11799] = 0, a [11800] = 5	5. 🗔				
	1		Minimal: a.count = 2, a [1] = 0, a [2] = 1.					

Fig. 2. Proof result in AutoProof: the first row (highlighted in red) indicates a proof failure; the second row is a counterexample generated based on the Z3 model; the third row is a minimized counterexample.

This minimized counterexample provides a simple diagnostic trace of max: on loop initialization at line 7, **Result** = 0 and i = 2; at line 13, the exit condition of the loop evaluates to **True** with account = 2 and i = 2, which forces the loop to terminate. These values reveal the fault in the program: the loop terminates too early, preventing the program from getting to the actual maximum value, found at position 2 of the array a. To eliminate this error, it suffices to strengthen the exit condition to permit one more loop iteration: change $i \ge a.count to i > a.count.$

III. TECHNOLOGY STACK

This section introduces technologies used in the present work, including language and prover.

Eiffel [15], [16] is an object-oriented programming language which natively supports Design-by-Contract [14]. An Eiffel program consists of a set of classes. A class represents a set of run-time objects characterized by the *features* applicable to them. Fig. 3 shows a simple class representing bank accounts. The class contains two types of features: attributes representing data items associated with instances of the class, such as balance (line 2) and credit_limit (line 4); routines representing operations applicable to these instances, including available_amount and transfer. Routines are further divided into procedures (with no returned value) and functions (returning a value). Here, available_amount is a function returning an integer (represented by the special variable **Result**), and transfer is a procedure.

1	class ACCOUNT feature			
2	balance: INTEGER			
3	Balance of this account.			
4	credit_limit: INTEGER			
5	Credit limit of this account.			
6	available_amount: INTEGER			
7	Amount available on this account.			
8	do			
9	Result := balance - credit_limit			
10	end			
11	transfer (amount: INTEGER; other: ACCOUNT)			
12	Transfer `amount' to the `other' account.			
13	require			
14	amount >= 0			
15	amount <= available_amount			
16	do			
17	balance := balance - amount			
18	other.balance := other.balance + amount			
19	ensure			
20	withdrawal: balance = old balance - amount			
21	deposit: other.balance = old other.balance			
	+ amount			
22	end			
23	end			

Fig. 3. A class implementing the behavior of bank accounts

Programmers can specify the properties of Eiffel classes by equipping them with contracts of the following types:

- A precondition (require) must be satisfied at the time of any call to the routine; the precondition of transfer (lines 13 - 15), for example, requires the value of amount to be non-negative an no greater than available_amount.
- A postcondition (ensure) must be guaranteed on routine's exit; for instance, a postcondition of transfer

at line 20 states that, at the end of the execution of transfer, the value of balance must have been decreased by amount.

- A loop invariant (invariant) characterizes the semantics of a loop in the form of a property satisfied after initialization and preserved by every iteration, as illustrated by the invariant of max (lines 9 - 11 in Fig. 1) specifies the properties of i and **Result** before and after every iteration.
- A loop variant (variant) is an integer measure that should always be non-negative and decrease strictly at each loop iteration, ensuring that the loop eventually terminates; the loop variant of the loop in max is a.count - i (line 19 in Fig. 1).

Contracts embedded in the code make it amenable to both dynamic analysis (run-time checking of the contract properties), as in the EiffelStudio environment, and static analysis (Hoare-style proofs), as in AutoProof.

AutoProof [1], [23] is a static verifier that checks the correctness of Eiffel programs against their functional specifications (contracts).

When verifying an Eiffel program, AutoProof translates the program into a Boogie program [2], [11], which is then transformed into a set of verification conditions (VCs) in SMT-LIB [3] format, based on Dijkstra's weakest precondition calculus [8]. The program's correctness follows from the validity of the VCs. Boogie asks an SMT solver (by default Z3, as noted) to reason about the validity of each VC. Specifically, the solver tries to find a model (an interpretation of variables and functions used in the SMT encodings) that satisfies the negation of a VC. If the solver is unable to find such a model (no counterexample exists and thus VC is a tautology), the verification is successful. If it succeeds in obtaining such a model, the verification fails and the solver makes the model available. This model witnesses the invalidity of the VC [12] and thus can be seen as a counterexample¹ at the SMT level. In general, an SMT model describes an execution trace (a sequence of program states) of a failed routine, along which the program goes to an error state. The CEAM approach makes use of such models to generate easy-to-understand counterexamples.

IV. COUNTEREXAMPLE EXTRACTION AND MINIMIZATION

This section first shows how to generate counterexamples based on SMT models, then presents the details of the CEAM strategy for counterexample minimization.

A. Counterexample extraction

In general, to construct a counterexample it suffices to extract the concrete values of relevant variables from the corresponding SMT model, and to use these values to produce a counterexample message (as in Fig. 2). The format of the message can vary depending on the chosen "verbosity level". As the goal of the approach and the tools is to ease the burden on programmers, the message only displays the initial values of relevant input variables in the counterexample, as illustrated in the case at the beginning of this article.

Fig. 4 shows a simplified portion of the Z3 model² corresponding to the failed proof of transfer's postcondition labeled by *withdrawal* (line 20 in Fig. 3). To construct a counterexample for this failure, it suffices to obtain the initial values of its three input variables: the implicit variable **Current**³ and the two arguments amount and other.

- 1 amount $\rightarrow 5799$
- 2 Heap $\rightarrow T@U!val!17$
- 3 Current \rightarrow T@U!val!18
- 4 other $\rightarrow T@U!val!18$
- 5 ACCOUNT.balance \rightarrow T@U!val!7
- 6 ACCOUNT.credit_limit \rightarrow T@U!val!8
- 7 Select \rightarrow {
- 8 T@U!val!17 T@U!val!18 T@U!val!7 →(-2147475928)
- 9 T@U!val!17 T@U!val!18 T@U!val!8 \rightarrow (-2147481727)

```
10 }
```

Fig. 4. A slice of model of the proof failure of withdrawal

In the transformation from Eiffel program to SMT code, to encode the execution semantics of an object-oriented program, the evolution of the *heap* (the collection of program objects) during an execution is modeled as a sequence of constants prefixed with Heap. Here the SMT constant Heap (line 2) corresponds to the heap at the initial program state of transfer. Select (line 7) is a function for retrieving the values of objects' fields. It takes three parameters, i.e., a heap state, an object reference and a data field, and returns the value of the specified field.

As the example shows, the concrete values of primitive variables (such as amount) are given directly, whereas the values of non-primitive variables (e.g., Current and other of ACCOUNT type) appear in a symbolic form, prefixed with T@U!val!. Such symbolic values can be seen as abstract memory locations for the corresponding variables (see [4], [13]). The Select function is available to obtain the values of the corresponding fields. For example, Current is an instance of ACCOUNT and thus has fields balance and credit_limit. The initial value of balance can be obtained by applying Select to a tuple made of Heap = T@U!val!17 (line 2), Current = T@U!val!18 (line 3) and ACCOUNT.balance = T@U!val!7 (line 5); the tuple matches the mapping in line 8, therefore the returned value for balance is -2147475928. Similarly, the value of credit_limit can be retrieved through the mapping in line 9.

To display the value of a non-primitive variable in the resulting counterexample message, the strategy first checks whether the variable has an alias that has been looked up

¹The counterexample is a *potential* counterexample since it can occasionally be spurious because of the prover's incompleteness, although that phenomenon is not significant in our experience.

 $^{^{2}}$ As the models are too big to include in their entirety, this presentation only displays the parts relevant to the discussion.

 $^{^3\}mbox{Current}$ represents the active object in the current execution context, similar to this in Java.

earlier; if so, it displays the alias relation between the variable and its earliest alias in the message; otherwise, it looks up all of its primitive data fields transitively and display them in the message.

In this example, the model shows that other and Current have the same symbolic values. Current is, consequently, an alias of other. As Current is processed prior to other, the fields of other will not be looked up and the message will display the alias relation between other and Current.

After applying the above rules, the counterexample can be derived: balance = -2147475928, credit_limit = -2147481727, amount = 5799, other = **Current**.

For variables of container types such as arrays or sequences, the resulting message displays the values of their sizes and containing elements. Here, we use the example of \max in Fig. 1 to demonstrate counterexample extraction for container types.

The AutoProof approach specifies container structures in terms of mathematical structures [23]. For example, the content of the input array a of max is specified through a special attribute sequence (see lines 10 - 11 in Fig. 1), which represents the mathematical sequence of integer values stored in a's cells. To obtain the content of a from the counterexample, we need to get the content of its sequence field from the model. Fig. 5 shows a slice of the model for the proof failure of max. CEAM first extracts the value of sequence by querying Select with the values of Heap, a, and ARRAY^INTEGER^.sequence (lines 1 - 3). Those values match to the mapping in line 5, hence the value of sequence is T@U!val!38. By using this value, CEAM then queries the two functions Seg#Length and Seg#Item to get the values of sequence's size (line 8) and elements (lines 12 - 14), respectively.

```
1 Heap →T@U!val!26
2 a →T@U!val!18
3 ARRAY^INTEGER_32^.sequence →T@U!val!9
4 Select →{
5 mout alog mout alog mout alog mout.
```

```
5 T@U!val!26 T@U!val!18 T@U!val!9 \rightarrow T@U!val!38
```

```
6 }
```

```
7 Seq#Length \rightarrow{
```

```
8 T@U!val!38 →11800
9 T@U!val!40 →28101
10 }
11 Seq#Item →{
12 T@U!val!38 1 →0
13 T@U!val!38 11799 →0
```

```
14 T@U!val!38 11800 \rightarrow 5
```

```
15 }
```

Fig. 5. A snippet of the model of proof failure of is_max

B. Counterexample minimization

Some of the extracted values found to cause a failure, such as -2147481727 in the above counterexample, are too large to enable a programmer to visualize easily the cause of the proof failure. CEAM can simplify counterexamples by reducing the absolute value of such integers. Program verification is

modular, meaning it processes each routine independently; so does minimization.

As the counterexample of a routine r consists of the initial states of its input variables, to minimize a counterexample of r it suffices to minimize each of r's input variables. The task of minimizing an input variable x can be reduced to a set of integer minimization tasks based on the type of x. The procedure minimize_integer in in Algorithm 1 minimizes an integer variable. It applies to the absolute value, retaining the sign (lines 1 - 5). If x is an object reference, the algorithm first checks whether x denotes a container; if yes, it finds the minimal size of x (lines 8 - 9) and then minimizes x's elements one by one (lines 11 - 13); otherwise it minimizes each of x's fields (lines 15 - 16).

Algorithm 1: minimize_general (x): minimize a	ı vari-
able of integer or reference type	

_						
1	if x is an integer then					
2	if $v > 0$ then					
3	minimize_integer (x)					
4	elseif $v < 0$ then					
5	minimize_integer $(-x)$					
6	elseif x is an object reference then					
7	if x is a container then					
8	minimize_integer (x.count)					
9	$n \leftarrow \text{minimized value of } x.count$					
10	from					
11	$ i \leftarrow 1$					
12	until					
13	$i \leq n$					
14	loop					
15	minimize_general $(x[i])$					
16						
17	else					
18	across each field f of x as $x \cdot f$ loop					
	minimize_general $(x.f)$					

Algorithm 2 shows the details of minimize_integer. B represents the Boogie procedure of routine r generated by Auto-Proof. The core idea is to find the smallest integer bound m such that adding a precondition $0 \le x < m$ (line 10) to B still yields the same verification results. When the algorithm ends (no smaller value of m can be found), the model from the last verification run is the minimal possible.

The algorithm starts by assigning to m the value of x in the initial model, then iteratively decreases m. At each iteration, it adds a new precondition to B to reduce the the range of x and performs verification based on the updated B, to check whether there still exists a model with a smaller value of x within the interval [0, m).

Picking a smaller value for m (line 9) can be implemented either by sequentially decreasing the value or using a binary reduction (as in binary search) for acceleration. The current implementation first checks whether x can be 0; if yes, the minimization stops as it has found the minimum; otherwise, **Algorithm 2:** minimize_integer (*x*): minimize an integer

1 from 2 $m \leftarrow \text{current value of } x$ 3 B.add_precondtion $(0 \le x \land x < m)$ verify 4 5 until no smaller value can be tried 6 7 loop B.remove_last_precondition 8 $m \leftarrow \text{pick}$ a smaller value 9 B.add_precondtion $(0 \le x \land x < m)$ 10 11 verify

it continues applying binary reduction. For more flexibility, it uses two user-specified parameters controlling termination:

- *Tolerance*: lower bound on the size of interval used in the binary search algorithm;
- *Max iteration*: maximum number of verification iterations allowed when minimizing an integer.

We have not endeavored to prove the correctness of the algorithms since the correctness of the approach (the "proof of the pudding") is embodied in the result: as the overall goal is to obtain a counterexample, the final criterion is whether the minimized value is still a counterexample, as established rigorously by the underlying proof technology. If not, the original unminimized counterexample still applies.

V. EXPERIMENT

A preliminary evaluation of the usability of the CEAM approach covers over 40 program versions⁴ of 9 examples, including some adapted from software verification competitions [5], [10], [24]. The examples (listed in Table I) include: 1) ACCOUNT introduced in Fig. 3; 2) a CLOCK class implementing a clock counting seconds, minutes, and hours; 3) a HEATER class implementing a heater adjusting it state (on or off) based on the current temperature and a user-defined temperature; 4) a LAMP class describing a lamp equipped with a switch (for switching on/off the lamp) and a dimmer (for adjusting the light intensity of the lamp); 5)a BINARY_SEARCH class implementing the binary search algorithm; 6) a LINEAR_SEARCH implementing the linear search algorithm; 7) a SQUARE_ROOT that calculates two approximate square roots of a positive integer; 8) MAX from Fig. 1; 9) a SUM_AND_MAX class computing the maximum and sum of the elements of an array.

Each row in the table reports on the experiment result of a single example, which consists of multiple versions. Each version was intentionally injected with a fault, such as confusions between + and -, \leq and <, > and \geq , missing loop invariant(s), pre- or postcondition, etc. The experiments use AutoProof to verify the programs, produce counterexamples for all occurring proof failures, and minimize them. The *tolerance* and *max iteration* parameters are currently set to 0 and 20 respectively.

The third column gives the total number of integer variables whose minimized in the experiment. Cases where no minimization is performed (e.g., the value of an integer variable in the counterexample is already 0 before minimization) are not included. The reduction rate (fourth column), number of iterations (fifth column), verification time (sixth column) and minimization time (last column) per integer are averaged out over all minimized integers of each example.

As the experiment result shows, CEAM minimization is cost-effective overall: in most cases, conducting minimization reduces the values of integer variables by over 80% with an average cost of less than 4 extra verification runs (iterations). Most of the minimized integer values are fairly small and easy-to-understand: out of 125 minimized values, 108 are in the range [-2, 2], out of which 58 are zero; values not in that range are usually close to the values of some predefined constants in the program.

VI. RELATED WORK

In line with the objective of helping programmers to understand the causes of proof failures, several approaches have been proposed to provide more user-friendly visualizations of counterexample models [6], [9], [11], [18], [22]. Claire et al. [11] developed the Boogie Verification Debugger (BVD), which interprets a counterexample model as a static execution trace (i.e., a sequence of abstract states). David et al. [9] transformed the models back into a counterexample trace comprehensible at the original source code level (SPARK) and display the trace using comments. Similarly, Aleksandar et al. [6] transformed SMT models to a format close to the Dafny syntax. In contrast to the present work, these approaches concentrate on the generation of human-readable counterexample and do not consider any counterexample minimization.

Another direction of work to ease the understanding of proof failures is to generate more useful counterexamples in the first place: Polikarpova et al. [21] developed a tool, Boogaloo, which applies symbolic execution to generate counterexamples for failed Boogie programs. Like the present approach, Boogaloo displays minimal counterexamples in the form of valuations of relevant variables. Müller et al. [17] implemented a Visual Studio dynamic debugger plug-in for Spec#, to reproduce a failing execution from the view of the prover. Likewise, Petiot et al. [19], [20] developed STADY, which produces failing tests for the failed assertions using symbolic execution techniques. That approach is also referred to as testing-based counterexample synthesis: it first translates the original C program into programs suitable for testing (runtime assertion checking), then applies symbolic execution to generate counterexamples (input for failing tests) based on the translated program. Unlike to that approach, CEAM counterexample extraction directly exploits the counterexample models produced by the provers, and hence does not require additional program instrumentation or counterexample generation.

⁴https://github.com/huangl223/Proof2Test/tree/main/examples

TABLE I					
EXPERIMENT RESULTS					

Example	Number	Total Number of	Avg. Reduction	Avg. Number	Avg. Verification	Avg. Minimization
Example	of versions	Minimized Integers	Rate	of Iterations	Time (seconds)	Time (seconds)
ACCOUNT	7	17	99.98%	2.5	0.028	0.087
CLOCK	6	13	100%	1.46	0.019	0.034
HEATER	2	4	48.4%	4.25	0.030	0.128
LAMP	4	8	0.819%	1.875	0.115	0.233
BINARY_SEARCH	5	31	98.8%	3.22	0.448	1.512
LINEAR_SEARCH	3	9	99.9%	3.44	0.087	0.279
SQUARE_ROOT	4	3	89.9%	4	0.133	0.505
MAX	4	12	87.1%	4.25	0.213	1.456
SUM_AND_MAX	6	11	80.7%	3.45	0.590	1.704

VII. CONCLUSION

This article has presented Counterexample Extraction and Minimization (CEAM), an approach that improves the quality of counterexamples generated in the presence of failed program proofs. CEAM automatically generates simple and easyto-understand counterexamples. We believe this makes the results of failed proofs practical enough to be used by regular programmers. The CEAM implementation is integrated in the AutoProof verifier to assist programmers when debugging failed proofs. The approach could also be applied to other Hoare-style verification tools relying on Boogie-style provers and SMT solvers.

Ongoing work includes implementing a feature of automatic test generation based on the counterexamples produced by CEAM, as well as extending the scope of CEAM to include the supports for more data types and program constructs. We also plan to conduct systematic empirical studies to evaluate precisely the benefits of the proposed techniques for programmers with no verification expertise.

Acknowledgments: We thank the anonymous referees for comments which led to significant improvements. The work benefitted from discussions with Filipp Mikoian, Alexander Kogtenkov and Alexander Naumchev from SIT.

References

- [1] AutoProof, http://comcom.csail.mit.edu/autoproof/
- [2] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387. Springer (2005)
- [3] Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB Standard: Version 2.0. In: International Workshop on Satisfiability Modulo Theories. vol. 13, p. 14 (2010)
- [4] Bjørner, N., Moura, L.d., Nachmanson, L., Wintersteiger, C.M.: Programming Z3. In: International Summer School on Engineering Trustworthy Software Systems. pp. 148–201. Springer (2018)
- [5] Bormer, T., Brockschmidt, M., Distefano, D., et al.: The COST IC0701 Verification Competition. In: International Conference on Formal Verification of Object-Oriented Software (FoVeOO). pp. 3–21. Springer (2011)
- [6] Chakarov, A., Fedchin, A., Rakamarić, Z., Rungta, N.: Better Counterexamples for Dafny. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 404–411. Springer (2022)

- [7] De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340. Springer (2008)
- [8] Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
- [9] Hauzar, D., Marché, C., Moy, Y.: Counterexamples from Proof Failures in SPARK. In: International Conference on Software Engineering and Formal Methods (SEFM). pp. 215–233. Springer (2016)
- [10] Klebanov, V., Müller, P., et al.: The 1st Verified Software Competition: Experience Report. In: International Symposium on Formal Methods (FM). pp. 154–168. Springer (2011)
- [11] Le Goues, C., Leino, K.R.M., Moskal, M.: The Boogie Verification Debugger. In: International Conference on Software Engineering and Formal Methods. pp. 407–414. Springer (2011)
- [12] Leino, K.R.M., Millstein, T., Saxe, J.B.: Generating Error Traces from Verification-Condition Counterexamples. Science of Computer Programming 55(1-3), 209–226 (2005)
- [13] Leino, K.R.M., Rümmer, P.: The Boogie 2 Type System: Design and Verification Condition Generation
- [14] Meyer, B.: Applying "Design by Contract". Computer 25(10), 40–51 (1992)
- [15] Meyer, B.: Object-Oriented Software Construction, vol. 2. Prentice Hall (1997)
- [16] Meyer, B.: Touch of Class: Learning to Program Well with Objects and Contracts. Springer (2016)
- [17] Müller, P., Ruskiewicz, J.N.: Using Debuggers to Understand Failed Verification Attempts. In: International Symposium on Formal Methods (FM). pp. 73–87. Springer (2011)
- [18] Nilizadeh, A., Calvo, M., Leavens, G.T., Cok, D.R.: Generating Counterexamples in the Form of Unit Tests from Hoare-style Verification Attempts. In: International Conference on Formal Methods in Software Engineering (FormaliSE). pp. 124–128. IEEE (2022)
- [19] Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your Proof Fails? Testing Helps to Find the Reason. In: International Conference on Tests and Proofs (TAP). pp. 130–150. Springer (2016)
- [20] Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: How Testing Helps to Diagnose Proof Failures. Formal Aspects of Computing (FAC) 30(6), 629–657 (2018)
- [21] Polikarpova, N., Furia, C.A., West, S.: To Run What No One Has Run Before: Executing an Intermediate Verification Language. In: International Conference on Runtime Verification (RV). pp. 251–268. Springer (2013)
- [22] Stoll, C.: SMT Models for Verification Debugging. Master thesis, ETH Zurich (2019)
- [23] Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: Autoproof: Auto-active Functional Verification of Object-Oriented Programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 566–580. Springer (2015)
- [24] Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental Benchmarks for Software Verification Tools and Techniques. In: Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE). pp. 84–98. Springer (2008)