View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

# Explaining Counterexamples with Giant-Step Assertion Checking

Benedikt Becker, Cláudio Lourenço, Claude Marché

## To cite this version:

Benedikt Becker, Cláudio Lourenço, Claude Marché. Explaining Counterexamples with Giant-Step Assertion Checking. F-IDE 2021 - 6th Workshop on Formal Integrated Development Environments, May 2021, Virtual, United States. hal-03217393

**HAL Id: hal-03217393**

**https://hal.inria.fr/hal-03217393**

Submitted on 4 May 2021

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Explaining Counterexamples with Giant-Step Assertion Checking[*]

Benedikt Becker Cláudio Belo Lourenço
Claude Marché
Université Paris-Saclay, CNRS, Inria, LMF, 91405, Orsay, France

Identifying the cause of a proof failure during deductive verification of programs is hard: it may be due to an incorrectness in the program, an incompleteness in the program annotations, or an incompleteness of the prover. The changes needed to resolve a proof failure depend on its category, but the prover cannot provide any help on the categorisation. When using an SMT solver to discharge a proof obligation, that solver can propose a model from a failed attempt, from which a possible counterexample can be derived. But the counterexample may be invalid, in which case it may add more confusion than help.

To check the validity of a counterexample and to categorise the proof failure, we propose the comparison between the run-time assertion-checking (RAC) executions under two different semantics, using the counterexample as an oracle. The first RAC execution follows the normal program semantics, and a violation of a program annotation indicates an incorrectness in the program. The second RAC execution follows a novel "giant-step" semantics that does not execute loops nor function calls but instead retrieves return values and values of modified variables from the oracle. A violation of the program annotations only observed under giant-step execution characterises an incompleteness of the program annotations. We implemented this approach in the Why3 platform for deductive program verification and evaluated it using examples from prior literature.

## 1 Introduction

Deductive program verification aims at checking that a given program respects a given functional behaviour. The expected behaviour is expressed formally by logical assertions, principally pre-conditions and post-conditions on procedures and functions, and invariants on loops. The verification process consists in generating, from the code and the formal annotations, a set of logic formulas called *verification conditions* (VCs), typically via a *Weakest Precondition Calculus* (WP) [8]. If the VCs are proved valid, then the program is guaranteed to satisfy its specification. Deductive verification environments like Dafny [10], OpenJML [6], or Why3 [4], pass the VCs to automatic theorem provers usually based on *Satisfiability Modulo Theories* (SMT), such as Alt-Ergo [3], CVC4 [1] and Z3 [12]. Due to the nature of these solvers, the conclusion of each VC is negated to form a proof goal and the background solver is queried for satisfiability. Since these solvers are assumed to be sound when the answer is "unsat", one can conclude that the VC is valid when that is indeed the answer.

In this paper we address the case when the solver does not answer "unsat", and provide a method to explain why the proof could not be completed. The solver may give instead several other answers: at best it answers "sat", possibly with a *model*, which is a collection of values for the variables in the goal. As displayed in Fig. 1, we rely on an approach by Dailler et al [7] to turn such a model into a *candidate counterexample*, which is essentially a collection of triples (variable, source location, value)

[*]This work was partly funded by collaborations between Inria and the companies AdaCore (Paris, France) and Mitsubishi Electric R&D Centre Europe (Rennes, France).

© Becker & Belo Lourenço & Marché
This work is licensed under the
Creative Commons Attribution License.

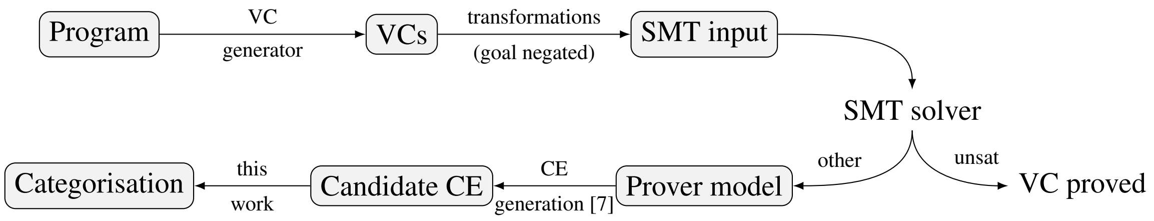

Figure 1: Pipeline of the counterexample generation and categorisation

representing the different values taken by a program variable during an execution. Such a counterexample may indicate two different problems: (i) a *non-conformance*, where the code does not satisfy one of its annotations; (ii) a *subcontract weakness*, where the annotations are not appropriate to achieve a proof (typically a weakness of a loop invariant or post-condition). Unfortunately there is no direct way to distinguish these cases. Other answers of the prover are even less informative: the answer “unknown” replaces “sat” when the solver is incomplete, for example in presence of quantified hypotheses or formulas involving non-linear arithmetic, or the solver reaches a time limit or a memory limit. In these cases, the solver may as well propose a model, but without any guarantee about its validity. Summing up, for any other answer than “unsat”, there is a need to validate the resulting candidate counterexample and categorise it as non-conformance or subcontract weakness.

We propose the categorisation of counterexamples using a novel notion of assertion checking, called *giant-step* assertion checking. Let us illustrate the idea on the following toy program, that operates on a global variable x.

```
let set_x (n:int) : unit ensures { x > n }
  x ← n + 1

let main () : unit
  x ← 0; set_x 2; assert { x = 3 }
```

The VC for the function `main` is $\forall x.\, x = 0 \rightarrow \forall x'.\, x' > 2 \rightarrow x' = 3$ where $x'$ denotes the new value of `x` after the call to `set_x`, and the premise of the second implication comes from the post-condition of `set_x`. The query sent to the solver is $x = 0 \wedge x' > 2 \wedge \neg(x' = 3)$, from which the solver would typically answer “sat” with the model $\{x = 0, x' = 4\}$, corresponding to a counterexample where `x` is 0 after the assignment and 4 after the call to `set_x`. If we proceed to a regular assertion-checking execution of `main`, no issue is reported: both the post-condition of `set_x` and the assertion in `main` are valid. Our giant-step assertion checking executes calls to sub-programs in a single step, selecting the values of modified variables from the candidate counterexample: it handles the call `set_x 2` by extracting the new value for `x` from the counterexample, here 4, and checking the post-condition, which is correct here. The execution then fails because the assertion is wrong. Since the standard assertion checking is OK but the giant-step one fails, we report a subcontract weakness. This is the expected categorisation, suggesting the user to improve the contract of `set_x`, in this case by stating a more precise post-condition. Giant-step assertion checking also executes loops by a single giant step to identify subcontract weaknesses in loop invariants.

In Section 2, we explain in more details the concept of giant-step execution, and how we use it to categorise counterexamples. In Section 3, we present our implementation, and experimental results, that were conducted within the Why3 environment [4]. We discuss related work and future work in Section 4. The technical details that we skip here due to lack of space are available in a research report [2].

```
evaluate(context,oracle,$e$):
 if expression $e$ is:
 • a function call (f $e_1$ ··· $e_n$):
     query the context for declaration of $f$:
       parameters $x_1 \cdots x_n$, pre-condition $\phi_{pre}$, writes $y_1 \cdots y_k$ post-condition $\phi_{post}$
     evaluate each $e_i$ to value $t_i$, and set each $x_i$ to $t_i$ in the context
     evaluate $\phi_{pre}$, if false report assertion failure
     query the oracle for values $v_1 \cdots v_k$ for $y_1 \cdots y_k$ and $v$ for result, assign them in the context
     evaluate $\phi_{post}$, if false report execution stuck
     return value $v$
 • a loop (while $c$ invariant { $\phi_{inv}$ } writes { $y_1 \cdots y_k$ } do $e$ done) :
     evaluate $\phi_{inv}$, if false report assertion failure
     query the oracle for values $v_1 \cdots v_k$ for $y_1 \cdots y_k$, assign in the context
     evaluate $\phi_{inv}$, if false report execution stuck
     evaluate $c$, if false return
     evaluate loop body $e$
     evaluate $\phi_{inv}$, if false report assertion failure
     report execution stuck
 • otherwise: as standard assertion-checking execution
```

Figure 2: Pseudo-code for giant-step assertion-checking execution

## 2 Giant-Step Assertion Checking

We consider here a basic programming language with mutable variables, function calls and while loops, where functions are annotated with contracts (a pre-condition and a post-condition) and loops are annotated with a loop invariant. The language also includes a standard `assert` statement, as shown in the example above.

**Giant-step assertion checking.** It corresponds to standard runtime assertion checking (RAC), in the sense that annotations (i.e., assertions, pre- and post-conditions, and loop invariants) are checked when they are encountered during execution. It differs from the standard RAC in the treatment of function calls and loops: instead of executing their body statements, they only involve a single, "giant" step. Giant-step execution is parameterised by an *oracle*, from which one retrieves the values of variables that are written by a function call or a loop. These written variables could be automatically inferred, but for simplicity we require them to be indicated by a `writes` clause. Therefore, a function declaration has the form:

```
let f ($x_1$: $\tau_1$) ··· ($x_n$: $\tau_n$) : $\tau$
  requires { $\phi_{pre}$ } ensures { $\phi_{post}$ } writes { $y_1,\ldots,y_k$ } = $e$
```

and a loop has the form:

```
while $e_1$ invariant { $\phi_{inv}$ } writes { $y_1,\ldots,y_k$ } do $e_2$ done
```

Figure 2 presents a pseudo-code for giant-step evaluation of a program expression $e$ (see [2] for a formal presentation). This execution form is inspired by the weakest precondition calculus, specifically

$$\begin{aligned}&\mathrm{WP}(\texttt{while}\ e_1\ \texttt{do invariant}\ \{\ \phi_{inv}\ \}\ \texttt{writes}\ \{\ y_1,\dots,y_k\ \}\ e_2\ \texttt{done}, Q) = \\ &\quad \phi_{inv}\ \wedge \forall v_1,\dots,v_k.\ (\phi_{inv} \rightarrow \mathrm{WP}(e_1, (\mathsf{result} \rightarrow \mathrm{WP}(e_2,\phi_{inv})) \wedge (\neg\mathsf{result} \rightarrow Q)))[y_j \leftarrow v_j]\end{aligned}$$

$$\mathrm{WP}((f\ t_1\ \cdots\ t_n), Q) = \phi_{pre}[x_i \leftarrow t_i]\ \wedge \forall v_1,\dots,v_k,v.\ (\phi_{post}[x_i \leftarrow t_i] \rightarrow Q)[y_j \leftarrow v_j][\mathsf{result} \leftarrow v]$$

where $v, v_1, \dots, v_k$ are fresh variables.

Figure 3: Rules for WP computation, loops and function calls

the rules for calls and loops that we remind in Figure 3. The execution may fail or get stuck in a number of situations:

- Line 7: if the pre-condition of a call to $f$ is false, the execution must fail (as in standard RAC).
- Line 9: if the post-condition of a call to $f$ is false, the values from the oracle are incompatible with the postcondition of $f$. A stuck execution is reported, and the counterexample will be declared invalid.
- Line 12: as in standard assertion checking, a failure is reported for the invariant initialisation.
- Line 14: if the invariant is false, the oracle does not provide valid values to continue the execution. A stuck execution is reported, and the counterexample will be invalid.
- Line 15: if the loop condition is false after setting the values of written variables in the context, the oracle covers an execution that goes beyond the loop, so we just terminate its execution.
- Line 17: if invariant is not preserved, we report an assertion failure.
- Line 18: if invariant is preserved, it means the oracle is not appropriate for identifying any failure, we report a stuck execution.

**Categorisation of counterexample.** As shown on Fig. 1, assume that a VC for a program is not validated by some SMT solver, which returns a model, which is turned into a candidate counterexample. We categorise this counterexample as follows (stopping when the first statement is met):

1. Run the standard RAC on the enclosing function of the VC, with arguments and values of global variables taken from the counterexample. If the result is
   (a) a failure at the same source location as the VC, we report a non-conformance: the code does not satisfy the corresponding annotation.
   (b) a failure at a different source location as the VC, the counterexample is bad (is not suitable to explain the failed proof), although it deserves to be reported as a non-conformance elsewhere in the code: it exposes an execution where the program does not satisfy some annotation.
2. Run the giant-step RAC on the enclosing function of the VC, with inputs and written variables given by the counterexample seen as an oracle. If the result is
   (a) a failure, we report a sub-contract weakness: some post-condition or loop invariant in the context is too weak, as in the introductory toy example.
   (b) a stuck execution, the counterexample is invalid and discarded: one of the premises of the VC is not satisfied, and we can suspect a prover incompleteness.
   (c) a normal execution, the counterexample is discarded: it satisfies the conclusion of the VC.

```
1   let isqrt (n: int)
2     requires { 0 <= n <= 10000 }
3     ensures { result * result <= n < (result + 1) * (result + 1) }
4   = let ref r = n in
5     let ref y = n * n in
6     let ref z = -2 * n + 1 in
7     while y > n do
8       invariant I1 { 0 <= r <= n }
9       invariant I2 { y = r * r }
10      invariant I3 { n < (r+1) * (r+1) }
11      invariant I4 { z = -2 * r + 1 }
12      y ← y + z; z ← z + 2; r ← r - 1
13    done;
14    r
```

Figure 4: Computation of the integer square root, adapted from C code of Petiot [13]

## 3 Experiments

We implemented our approach in the Why3 platform for deductive program verification, and tested it by reproducing the experiments from prior literature about the categorisation of proof failures (Petiot et al. [13]). These experiments covered three example programs written in C with ACSL program annotations, and the verification was carried out in the Frama-C framework. The experiments comprised modifications to the programs that introduced proof failures, which were then categorised using their approach. We translated the C programs to WhyML and were able to reproduce the categorisations with our approach for all 16 modifications that were applicable to the WhyML program.

The experiments by Dailler et al. [7] were written in Ada/SPARK, which uses Why3 for deductive verification. Their "Riposte" testsuite contains 24 programs with 247 checks. The integration of our approach with SPARK is work in progress. Currently, we are able to identify in this testsuite 14 wrong counterexamples, 57 non-conformities, and 22 other cases classified as "either non-conformity or subcontract-weaknesses" (an imprecise answer which results when the standard RAC is non-conclusive [2]). The counterexamples of 154 checks could not yet be categorised.

Let us illustrate our experiments on one of Petiot's examples [13] in C, a function that calculates the integer square root. Our translation of the C program to WhyML is shown in Figure 4. The result for parameter $n$ is an integer $r$ such that $r^2 \leq n < (r+1)^2$. The variable $r$ is initialised by $n$ as an over-approximation of the result, the variable $y$ by $n^2$, and $z$ by $-2*n+1$. During execution of the while loop, the value of $r$ is decremented and the value of $y$ is kept at $r^2$, while maintaining $n < (r+1)^2$. When the loop condition becomes false, $r$ contains the largest integer such that $r^2 \leq n$. The VC for function `isqrt` is split by Why3 into nine verification goals: two for the initialisation and preservation of each of the four loop invariant, and one for the post-condition. The validity of the program is proven in Why3 (we used the Z3 solver to prove the goals and generate counterexamples).

We reproduce here two variations of that program, that introduce proof failures. First, changing the first assignment in line 12 to `y ← y - z` leads to a proof failure for the preservation of invariant $I_2$. The counterexample gives the value 4 for the argument `n` of function `isqrt`. The standard RAC execution fails after the first loop iteration when checking the preservation of invariant $I_2$, and the proof failure is categorised as non-conformity. Second, removing loop invariant $I_3$ leads to a proof failure for the postcondition. The counterexample gives the value 1 for the argument `n`. The standard RAC execution terminates normally with the correct result of 1. The giant-step RAC execution initialises the variables `r`, `y`, and `z` with values 1, 1, and -1, respectively, and checks that the loop invariants hold initially. To execute the loop in a giant step, the variables `r`, `y`, and `z` are set to the values 0, 0, and 1 from the counterexample, which also satisfy the loop invariants. The loop condition becomes false, and the giant-step execution leaves the loop. The execution then fails because the current value 0 of variable `r` contradicts the postcondition. Since standard RAC terminated normally but giant-step RAC failed, the proof failure is categorised as a subcontract-weakness.

See the research report [2] for more examples and experimental results.

## 4 Related Work and Perspectives

Christakis et al. [5] use Dynamic Symbolic Execution (DSE) (i.e., concolic testing) to generate test cases for the part of the code that underlies a failing VC. The code is extended by run-time checks and fed to the DSE engine Delfy, that explores all possible paths up to a given bound. The output can be one of the following: the engine verifies the method, indicating that the VC is valid and thus no further action is required; it generates a test case that leads to an assertion violation, indicating that the VC is invalid; or no test case can be generated in the given bound, where nothing can be concluded. Our approach is more directly based on the work of Petiot et al. [13]. Their work also relies on a DSE engine (the PathCrawler tool) to classify proof failures and to generate counterexamples. Each proof failure is classified as *non-compliance*, *subcontract weakness*, or *prover incapacity*. By using DSE, every generated counterexample leads to an assertion failure during concrete execution of the program.

What distinguishes our approach from the two above is that we derive the test case leading to a proof failure from the model generated by the SMT solver, rather than relying on a separate tool such as the DSE engine. Instead of applying different program transformations, we compare the results of two different types executions (standard RAC and giant-step RAC) of the original program.

There are quite a lot of potential improvements and extensions to our work, which are discussed in our research report [2]. First, our approach should be extended to support more features such as the maintenance of type invariants. Our implementation deserves to be made more robust, to deal with missing values in the counterexample, and calls to functions that lack an implementation. A representative example of remaining technical issues is when the original code makes use of polymorphic data types: in that case, a complex encoding is applied to the VCs before sending them to provers, and unwinding this encoding to a obtain an oracle for running the RAC does not currently work. As seen in the experiments so far, our method is often limited in the RAC, for example when checking assertions that are quantified formulas, which is a known issue for RAC [9]. As mentioned in the experiments, our implementation aims to support various Why3 front-ends, such the one for Ada/SPARK [11]. The current experimental results are not fully satisfying and there are numerous required practical improvement. Yet, the ability to filter out obviously wrong counterexamples, and categorise them, is hopefully a major expected improvement for the SPARK user experience. A remaining challenge is that a counterexample at Why3's level might not be suitable at the front-end level. In particular, performing the small-step assertion checking in the front-end language may result in a more accurate diagnose.

**Acknowledgements.** We thank our partners from AdaCore (Yannick Moy) and Mitsubishi Electric R&D Centre Europe (David Mentré, Denis Cousineau, Florian Faissole and Benoît Boyer) for their

feedback on first experimentations of our counterexample categorisation approach.

## References

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In: *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, Springer-Verlag, Berlin, Heidelberg, pp. 171–177. Available at `http://dl.acm.org/citation.cfm?id=2032305.2032319`.

[2] Benedikt Becker, Cláudio Belo Lourenço & Claude Marché (2021): *Giant-step Semantics for the Categorisation of Counterexamples*. Research Report RR-9407, Inria. Available at `https://hal.inria.fr/hal-03213438`.

[3] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer & Alain Mebsout (2008): *The Alt-Ergo Automated Theorem Prover*. `http://alt-ergo.lri.fr/`.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2015): *Let's Verify This with Why3*. *International Journal on Software Tools for Technology Transfer (STTT)* 17(6), pp. 709–727, doi:10.1007/s10009-014-0314-5. Available at `http://hal.inria.fr/hal-00967132/en`. See also `http://toccata.lri.fr/gallery/fm2012comp.en.html`.

[5] Maria Christakis, K. Rustan M. Leino, Peter Müller & Valentin Wüstholz (2016): *Integrated Environment for Diagnosing Verification Errors*. In: *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*, Springer, pp. 424–441.

[6] David R. Cok (2014): *OpenJML: Software Verification for Java 7 using JML, OpenJDK, and Eclipse*. In Catherine Dubois, Dimitra Giannakopoulou & Dominique Méry, editors: *Proceedings 1st Workshop on Formal Integrated Development Environment*, *EPTCS* 149, pp. 79–92.

[7] Sylvain Dailler, David Hauzar, Claude Marché & Yannick Moy (2018): *Instrumenting a Weakest Precondition Calculus for Counterexample Generation*. *Journal of Logical and Algebraic Methods in Programming* 99, pp. 97–113. Available at `https://hal.inria.fr/hal-01802488`.

[8] Edsger W. Dijkstra (1976): *A discipline of programming*. Series in Automatic Computation, Prentice Hall Int.

[9] Nikolai Kosmatov, Claude Marché, Yannick Moy & Julien Signoles (2016): *Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014*. In Tiziana Margaria & Bernhard Steffen, editors: *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, *Lecture Notes in Computer Science* 9952, Springer, Corfu, Greece, pp. 461–478. Available at `https://hal.inria.fr/hal-01344110`.

[10] K. Rustan M. Leino & Valentin Wüstholz (2014): *The Dafny Integrated Development Environment*. In Catherine Dubois, Dimitra Giannakopoulou & Dominique Méry, editors: *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, *Electronic Proceedings in Theoretical Computer Science* 149, pp. 3–15, doi:10.4204/EPTCS.149.2.

[11] John W. McCormick & Peter C. Chapin (2015): *Building High Integrity Applications with SPARK*. Cambridge University Press.

[12] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3, An Efficient SMT Solver*. In: *TACAS*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340.

[13] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti & Jacques Julliand (2018): *How testing helps to diagnose proof failures*. *Formal Aspects Comput.* 30(6), pp. 629–657, doi:10.1007/s00165-018-0456-4. Available at `https://doi.org/10.1007/s00165-018-0456-4`.