Giant-step Semantics for the Categorisation of Counterexamples

Benedikt Becker, Cláudio Belo Lourenço, Claude Marché

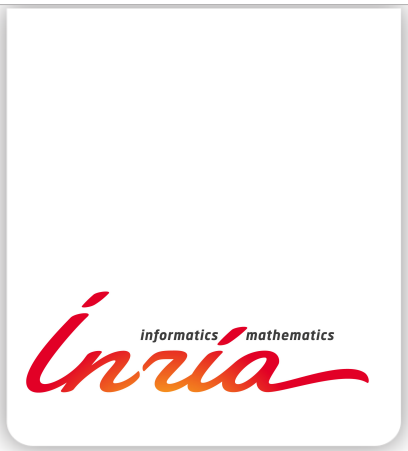
RESEARCH REPORT

N° 9407

April 2021

Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--9407--FR+ENG



Giant-step Semantics for the Categorisation of Counterexamples*

Benedikt Becker†, Cláudio Belo Lourenço†, Claude Marché†

Project-Team Toccata

Research Report n° 9407 — April 2021 — 43 pages

* This work has been partially supported by the bilateral contract ProofInUse-AdaCore between Inria team Toccata and AdaCore company, Paris, France ; and by the bilateral contract ProofInUse-MERCE between Toccata and Mitsubishi Electric R&D Centre Europe, Rennes, France.

† Université Paris-Saclay, CNRS, Inria, LMF, 91405, Orsay, France

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Abstract: Deductive Verification aims at verifying that a given program code conforms to a formal specification of its intended behaviour. That approach proceeds by generating mathematical statements whose validity entails the conformance of the program. Such statements are typically given to automated theorem provers. This work aims at providing help to the programmer in the case when the proofs fail.
Indeed, identifying the cause of a proof failure during deductive verification of a program is hard: it may be due either to an incorrectness in the program, to an incompleteness in the program annotations, or to an incompleteness of the prover itself. The kind of modifications to perform so as to fix the proof failure depends on its category, but the prover alone cannot provide any help on the categorisation.
In practice, when using an SMT solver to discharge a proof obligation, such a solver can propose a *model* from a failed proof attempt, from which a candidate *counterexample* can be derived. But it appears that such a counterexample may be invalid, in which case it may add more confusion than help to the programmer.
The goal of this work is both to check for the validity of a counterexample and to categorise the proof failure.
We propose an approach that builds upon the comparison between the run-time assertion checking (RAC) executions of the program (and its specifications) under two different semantics, using the counterexample as an oracle. The first RAC execution follows the normal program semantics, so that 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.

Key-words: Formal Specification, Deductive Verification, Generation of counterexamples, Runtime assertion checking, Program Verifier Why3

Une sémantique à pas de géant pour la catégorisation des contre-exemples[3]

Résumé : La vérification déductive cherche à vérifier que le code d'un programme donné est conforme à une spécification formelle de son comportement prévu. Cette approche procède en générant des énoncés mathématiques dont la validité implique la conformité du programme. De tels énoncés sont généralement transmis à des prouveurs automatiques de théorèmes. Le travail présenté dans ce rapport vise à fournir une aide au programmeur dans le cas où les preuves échouent.

En effet, identifier la cause d'un échec de preuve lors de la vérification déductive d'un programme est difficile: cela peut être dû soit à une erreur dans le programme, soit à une incomplétude dans les annotations du programme, ou encore à une incomplétude du prouveur lui-même. Le type de modification à effectuer pour corriger l'échec de la preuve dépend de sa catégorie, mais le prouveur ne peut à lui seul fournir de l'aide à cette catégorisation.

En pratique, lors de l'utilisation d'un solveur de type SMT pour prouver une obligation de preuve, un tel solveur peut proposer un *modèle* à partir d'une tentative de preuve ratée, à partir duquel un *contre-exemple* potentiel peut être dérivé. Mais il s'avère qu'un tel contre-exemple peut être invalide, auquel cas il peut apporter plus de confusion que d'aide au programmeur.

Le but de ce travail est à la fois de vérifier la validité d'un contre-exemple et de catégoriser l'échec de la preuve. Nous proposons une approche qui s'appuie sur la comparaison entre des exécutions « RAC » (*Run-time Assertion Checking*) du programme (et de ses spécifications) sous deux sémantiques différentes, en utilisant le contre-exemple comme un oracle. La première exécution RAC suit la sémantique habituelle du programme, de sorte qu'une violation d'une annotation du code indique une erreur dans le programme. La deuxième exécution RAC suit une sémantique originale dite « à pas de géant » qui n'exécute ni les boucles ni les appels de fonction, mais récupère à la place depuis l'oracle les valeurs de retour et les valeurs des variables modifiées. Une violation des annotations du programme observée uniquement sous l'exécution à pas de géant caractérise une incomplétude des annotations du programme.

Nous avons implémenté cette approche dans la plateforme Why3 pour la vérification déductive, et nous l'avons évalué sur une base d'exemples tirés de la littérature antérieure.

Mots-clés : Spécification formelle, preuve de programmes, génération de contre-exemples, vérification d'assertions à l'exécution, environnement de preuve Why3

Contents

1 Context, Motivations and Informal Introduction to the Approach 6
1.1 Classification of Programs Errors, Informally . . . 7
1.2 Overview of this report . . . 8

2 A Core Language with a Concrete Execution Semantics and a Calculus of Verification Conditions 9
2.1 The μWhy language . . . 9
2.2 Well-Typing of μWhy programs . . . 10
2.3 Concrete, Assertion-Checking Semantics . . . 11
2.3.1 Rules for Basic Expressions . . . 11
2.3.2 Execution of loops and function calls . . . 13
2.3.3 Explicitating Blocked Executions . . . 13
2.4 Generation of Verification Conditions . . . 15
2.5 Generating Candidate Counterexamples from Proof Failures . . . 18

3 Giant-Step Assertion-Checking 20
3.1 Giant-Step Assertion-Checking Semantics . . . 20
3.2 Automatic Classification of Counterexamples . . . 23

4 Extensions and Implementation 25
4.1 Refering to the past with labels . . . 25
4.2 Verification of Termination . . . 26
4.3 Dealing with "for" loops . . . 27
4.4 Why3 Implementation of Assertion Checking . . . 30
4.5 Why3 Implementation of CE generation and Categorisation . . . 31

5 Experimental Evaluation 33
5.1 Experiment: Integer square root . . . 33
5.2 Experiment: Binary search . . . 35
5.3 Experiment: Restricted growth . . . 37

6 Conclusions, Discussions, Related Work and Future Work 40
6.1 Related Work . . . 40
6.2 Future Work . . . 40

List of Figures

1 Syntax of the μWhy language . . . 9
2 Small-step operational semantics: basic expressions . . . 12
3 Small-step operational semantics: context rules . . . 12
4 Small-step operational semantics: loops . . . 12
5 Small-step operational semantics: function calls . . . 13
6 Small-step operational semantics: explicit blocking rules . . . 14
7 Rules for WP computation, basic expressions . . . 16
8 Rules for WP computation, loops and function calls . . . 16
9 Giant-step operational semantics: Function calls . . . 20
10 Giant-step operational semantics: loops . . . 21
11 Categorisation of candidate counterexamples . . . 23
12 Operational semantics for labels . . . 26
13 Small-step operational semantics: variant clauses . . . 26
14 Rules for WP computation, loops and function calls with variants . . . 27
15 Rules for giant-step execution, checking of variants . . . 27
16 Small-step operational semantics: for-loops . . . 28
17 Rules for WP computation of for-loops . . . 28
18 Giant-step operational semantics: for-loops . . . 29
19 Procedure for checking assertions during execution . . . 31
20 Pipeline of the Why3 counterexample (CE) generation and categorisation . . . 32
21 Computation of the integer square root in WhyML . . . 34
22 Modifications to the program `isqrt` . . . 34
23 Binary search in WhyML . . . 36
24 Modifications to the program `binary_search` . . . 36
25 Restricted growth for arrays in WhyML . . . 38
26 Modifications to the program implementing restricted growth . . . 39

Giant steps are what you take
Proving on the moon
I hope my code don't break
Proving on the moon
— Freely inspired from a song of the 20th century [19]

1 Context, Motivations and Informal Introduction to the Approach

The goal of deductive program verification is to check that a given program follows a given functional behaviour. In this context, the expected behaviour of the program must be expressed formally using logical annotations. Such annotations include assertions, pre-conditions and post-conditions on procedures and functions, and invariants on loops. The verification process is based on a set of logic formulas called *verification conditions* (VCs), which are generated from the code and its annotations typically using the *Weakest Precondition Calculus* (WP) [10]. If the VCs are valid, then the program is guaranteed to satisfy its specifications.

There are several mature deductive verification environments, such as Dafny [14], OpenJML [6] or Why3 [4]. In these environments, the generated VCs are passed on to automated theorem provers so as to be proved as valid theorems. The most commonly used provers in this context are those based on *Satisfiability Modulo Theories* (SMT). Among the provers of this kind, the preferred ones are those which supports quantified formulas, such as Alt-Ergo [3], CVC4 [1] and Z3 [9]. The primary purpose of SMT solvers is to take a set of logical assertions as input, and decide if they are satisfiable or not. In order to test the validity of a VC using an SMT solver, the conclusion of the VC is negated ($P \rightarrow Q$ is valid if and only if $P \wedge \neg Q$ is unsatisfiable). If the SMT solver then responds that the resulting set of formulas is unsatisfiable, then the VC is valid. SMT solvers are supposedly sound, *i.e.*, when they respond 'unsat' for a VC then that VC is valid.

In this report, we address the cases where the SMT solver does not answer "unsat", and we provide a method to explain why the proof could not be completed. The SMT solver may give several other kind of answers: at best it answers "sat", possibly with a *model*, which is a collection of values for the variables in the goal that can be turned into a *counterexample* for the annotated program [8]. A counterexample can indicate two different problems:

- a *non-conformance*, where the code does not satisfy one of its annotations; or
- 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 SMT solver are even less informative: the response "unknown" replaces "sat" when the SMT solver is incomplete, for example in presence of quantified hypotheses or formulas involving non-linear arithmetic, or when the SMT solver can reach a given time limit or a given memory limit. In all these cases, the SMT 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 counterexample obtained from the proposed model and categorise it as non-conformance or subcontract weakness.

1.1 Classification of Programs Errors, Informally

We propose the categorisation of counterexamples using a novel notion of run-time assertion checking (RAC). Let us illustrate the idea on the following example program, that operates on a global variable x.

```
1  fun set_x (n:int) : unit
2    ensures { x > n }
3    =
4    x ← n + 1
5
6  fun main_set () : unit
7    =
8    x ← 0;
9    set_x 2;
10   assert { x = 3 }
```

The VC for the function `main_set` 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 SMT solver is

$$x = 0 \wedge x' > 2 \wedge \neg(x' = 3)$$

to which the SMT solver typically answers "sat" with the model $\{x = 0, x' = 4\}$. If we proceed to a regular assertion-checking execution of that code, no issue is reported: both the post-condition of `set_x` and the assertion in `main_set` are valid. Our proposed variant of assertion checking, which we call *giant-step* assertion checking, executes calls to sub-programs in a single step, selecting the values of modified variables from the proposed model. This giant-step assertion checking, on the `main_set` function of the example above, will handle the call `set_x 2` by extracting the new value for x from the model, here 4, and checking the post-condition, which is correct. The execution then fails because the assertion is wrong. Since standard assertion checking is OK but giant-step execution fails, we report a subcontract weakness. This is the expected categorisation, suggesting to the user to improve the contract of `set_x`, in this case by stating a more precise post-condition. An adequate post-condition could be for example `ensures { x = n + 1 }`.

Giant-step assertion checking also executes loops by a single giant step to identify subcontract weaknesses in loop invariants. Here is an example program with a loop and an invariant:

```
1  fun main_loop () : unit
2    =
3    x ← 0;
4    while x < 10 do
5      invariant { x >= 0 }
6      x ← x + 1
7    done;
8    assert { x = 10 }
```

The VC for the function `main_loop` is:

$$\begin{array}{l}\forall x.\ x = 0\ \rightarrow \\ \quad x \geq 0 \wedge \\ \quad \forall x'.\ x' \geq 0 \rightarrow \texttt{if}\ x' < 10\ \texttt{then}\ x' + 1 \geq 0\ \texttt{else}\ x' = 10.\end{array}$$

The first line in the VC corresponds to the initial assignment, the second line to the initialisation of the loop invariant at loop entrance, and the third line correspond both to the preservation of the loop invariant (`then` branch) and the assertion of the code (`else` branch). It is indeed common in practice to split such a VC into several sub-goals, here

$$\forall x.\ x = 0\ \rightarrow x \geq 0$$

for the loop invariant initialisation,

$$\forall x.\ x = 0\ \rightarrow \forall x'.\ x' \geq 0 \wedge x' < 10 \rightarrow x' \geq 0$$

for the loop invariant preservation, and

$$\forall x.\ x = 0\ \rightarrow \forall x'.\ x' \geq 0 \wedge x' \geq 10 \rightarrow x' = 10$$

for the assertion. The two first sub-goals will be validated by an SMT solver. For the third sub-goal, however, which is passed to the SMT solver with negated conclusion as

$$x = 0 \wedge x' \geq 0 \wedge x' \geq 10 \wedge x' \neq 10,$$

the SMT solver will typically answer "sat" with a model like $\{x = 0, x' = 11\}$. On the one hand, a standard execution of program `main` will not reveal any non-conformance. On the other hand, the giant-step execution will (1) check the loop invariant for the initial value 0 for `x`, which is OK; (2) using the value 11 from the model it will check the loop invariant, which is again OK; (3) the loop condition evaluates to false, telling that the execution should proceed to what comes after the loop, and thus it will check the assertion in line 8 which is wrong. Again, a sub-contract weakness is reported, meaning that the loop invariant must be strengthened to prove the program. An appropriate loop invariant would be for example `invariant { 0 <= x <= 10 }`.

1.2 Overview of this report

In the rest of this document, we first introduce a core language, called μWhy, on which we will describe our approach (Section 2). We formally define an operational semantics that includes the checking of annotations, and we present a calculus to compute verification conditions. We then formalise the concept of giant-step execution (Section 3), and explain how we combine concrete execution and giant-step execution to categorise counterexamples (Section 3.2). Section 4 is dedicated to our implementation in the Why3 environment for deductive program verification. We also present a few extensions useful in practice, such as regarding the proof of termination. In Section 5 we report on our replication of experiments about the categorisation of proof failures from previous literature. Finally, we discuss related work and future work in Section 6.

p	::=	$d_1 \cdots d_n$	program
d	::=	`var` x `:` τ	global variable declaration
	|	`fun` f $(x_1 : \tau_1)$ $\ldots$ $(x_n : \tau_n)$ `:` τ	function declaration
		`requires {` ϕ_{pre} `} ensures {` ϕ_{post} `} writes {` $y_1, \ldots, y_k$ `} =` e	
τ	::=	`bool` | `int` | `unit`	type
ϕ	::=	$\top \mid \bot \mid t_1\ op\ t_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \to \phi_2 \mid$	
		$\forall x : \tau.\phi \mid \exists x : \tau.\phi$	formula
t	::=	$l \mid x \mid t_1\ op\ t_2$	pure term
l	::=	`()` | `true` | `false` | `0` | `1` | $\cdots$	literal
e	::=	t	pure expression
	|	$x \leftarrow e$	assignment
	|	`var` $x : \tau$ `=` e_1 `in` e_2	local binding
	|	`if` e_1 `then` e_2 `else` e_3	conditional
	|	`assert {` ϕ `}`	assertion
	|	`stuck`	diverging statement
	|	`while` e_1 `do invariant {` ϕ_{inv} `} writes {` $y_1, \ldots, y_k$ `}` e_2 `done`	loop
	|	$f\ x_1\ \cdots\ x_n$	function call

Figure 1: Syntax of the μWhy language

2 A Core Language with a Concrete Execution Semantics and a Calculus of Verification Conditions

In this section, we introduce a core language called μWhy, which is representative of the constructions we want to support. We first introduce the syntax and its informal meaning, then describe the formal operational semantics of the language and its annotations, and finally give the rules for computing the verification conditions.

2.1 The μWhy language

μWhy is a functional language with a ML-like syntax, extended with *mutable* variables, which are modifiable in-place. The grammar is shown in Figure 1. A μWhy program is a sequence of top-level declarations of mutable variables and functions. The data-types considered are Booleans (type `bool`), unbounded integers (type `int`), and the constant `()` of type `unit`.

A formula is composed by the typical logical connectors and relational operators over pure terms; on the other hand, pure terms are composed by literals, variables an pure binary operators over terms. We do not define precisely *op* but it denotes pure operators on integers and Booleans like $+$, $\times$ or $\leq$.

The body of functions is an expression, which may have side-effects as opposed to pure terms. There is no need for a separate notion of "statements": these are just expressions of type `unit`. Expressions extend pure terms with assignment, local binding, conditional, assert and stuck *annotations* while loops, and function calls. We use identifiers $x, y, \ldots$ to denote variables, $f, g, \ldots$ for function names.

Beside these standard language features, μWhy has explicit program annotations. A first kind is the `assert { }` expression, which checks that the formula ϕ holds: it returns `()` if this is the case or it blocks execution otherwise (assertion failure). The `stuck` expression can be seen as a statement that never returns – its usage will be clarified later. A while loop is annotated by a *loop invariant*, i.e. a formula that is intended to hold before entering the loop, and to remain true after each iteration. Moreover, loops

are given a `writes` clause that enumerates the mutable variables potentially modified in the loop body. Function declaration are annotated by a pre-condition, introduced by `requires`, and by a post-condition, introduced by `ensures`. Pre-conditions and post-conditions may refer to global variables and function parameters. The post-condition may also refer to the value returned by the function, using the keyword result. Variables that may be modified by a function have to be declared in the function's `writes` clause.

For simplicity we assume that arguments of function calls are variables, which is not a limitation since function arguments can always be bound to local variables before a call. Moreover, μWhy supports arbitrary recursive of mutually recursive calls between the functions declared in a program.

2.2 Well-Typing of μWhy programs

The μWhy programs we consider are assumed to be well-typed in a standard way. To ease the presentation of semantics rules later on, we also assume that there are no name clashes between global and local variables.

The `writes` clauses of function declarations and loops are also checked: for each assignment expression $x \leftarrow e$, in addition to check that x is declared and of the same type as e, the typing also checks that x is listed as a written variable in the function or loop that encloses it. Function calls are checked in a similar way: in a call to a function f the variables declared as written by the contract of f must be included in the set of written variables of the function or loop that encloses it. Similar, the variables declared as written by a loop must a subset of the variables declared written by the function or loop that encloses it. In fact, the `writes` clauses could be inferred automatically, with a computation of a fix-point. Yet, it is easier to assume that `writes` clauses are given and checked. The following two examples are well-typed including regarding the validity of the `writes` clauses.

Example 2.1 *The following is mostly the first example of the introducing section, where we introduce an extra local variable to handle the call to `set_x` inside the body of `main`, and where we make explicit the variables written by functions.*

```
1  var x : int
2
3  fun set_x (n:int) : unit
4    writes { x }
5    ensures { x > n }
6    =
7    x ← n + 1
8
9  fun main () : unit
10   writes { x }
11   =
12   x ← 0;
13   var temp : int = 2 in set_x temp;
14   assert { x = 3 }
```

Example 2.2 *The following is mostly the second example of the introducing section, where we make explict the variable written by the loop.*

```
1  var x : int
2
3  fun main () : unit
4    writes { x }
```

```
5    =
6    x ← 0;
7    while x < 10 do
8      invariant { x >= 0 }
9      writes { x }
10     x ← x + 1
11   done;
12   assert { x = 10 }
```

Another important typing condition that we assume here is in function calls: on any call $(f\ z_1\ \cdots\ z_k)$, the variables z_i must be distinct, and different from the variables $\vec{y}$ declared in the writes clause for f. This is a non-aliasing condition that is necessary for soundness of the VC generation described below in Section 2.4.

2.3 Concrete, Assertion-Checking Semantics

We define the operational semantics of μWhy programs using a classical small-step approach. This semantics includes the checking of annotations, hence formally defines the run-time assertion checking of programs. It is defined by judgements of the form $\Gamma_1, \Pi_1, e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_2$, meaning informally that in context (Γ_1, Π_1), the expression e_1 reduces in one step to the context (Γ_2, Π_2) and the next expression to evaluate is e_2. Γ denotes a value environment for the global variables and Π denotes a value environment for the local variables. In the semantic judgements below, the letter v denotes a value, which corresponds to a literal in Figure 1. Values are not reducible by the semantic judgements and represent terminated executions. If an expression is reduced to a value in a finite or infinite number of steps, we say that the expression *executes safely*. Besides a safe execution, an execution may only stop at a program annotation. The ultimate goal for program verification is then to show that for a given program, all executions are safe. We talk here about partial correctness: the question of proving termination will be addressed in Section 4.2.

2.3.1 Rules for Basic Expressions

Figure 2 introduces a first set of rules dedicated to basic expressions, i.e., all expressions except loops and function calls. We implicitly assume predefined rules to reduce the built-in operators.

Notice that an assertion is reduced to `()` only if the given formula is valid in the current context; if the assertion is violated the execution is blocked.

The idea of the `stuck` is the same as the *de facto* "`assume false`" annotation. An `assume` states a property that should be assumed instead of proved as with `assert`. In a formal semantics, "`assume P`" should be seen as a never returning statement when `P` does not hold [2]. As a consequence, not giving any rule for `stuck` could be seen as a mistake, we should have introduced a rule like

$$\frac{}{\Gamma, \Pi, \texttt{stuck} \rightsquigarrow \Gamma, \Pi, \texttt{stuck}}$$

which captures the intended meaning of `stuck`: a statement that never returns. However, in our case we decided not to give such a rule because it would prevent the specific treatment that we want to apply later in Section 2.3.3.

The initial set of rules for basic expressions are complemented with a set of so-called contextual rules in Figure 3, which define the reduction of sub-expressions.

$$\textsc{Global-Variable}\ \frac{\Gamma(x) = v}{\Gamma, \Pi, x \rightsquigarrow \Gamma, \Pi, v} \qquad \textsc{Local-Variable}\ \frac{\Pi(x) = v}{\Gamma, \Pi, x \rightsquigarrow \Gamma, \Pi, v}$$

$$\textsc{Local-Variable-binding}\ \frac{}{\Gamma, \Pi, \texttt{var}\ x = v\ \texttt{in}\ e \rightsquigarrow \Gamma, (x, v) \cdot \Pi, e}$$

$$\textsc{Global-Variable-assignment}\ \frac{x \in dom(\Gamma)}{\Gamma, \Pi, x \leftarrow v \rightsquigarrow \Gamma[x \leftarrow v], \Pi, ()} \qquad \textsc{Local-Variable-assignment}\ \frac{x \in dom(\Pi)}{\Gamma, \Pi, x \leftarrow v \rightsquigarrow \Gamma, \Pi[x \leftarrow v], ()}$$

$$\textsc{Conditional-true}\ \frac{}{\Gamma, \Pi, \texttt{if true then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \Gamma, \Pi, e_2} \qquad \textsc{Conditional-false}\ \frac{}{\Gamma, \Pi, \texttt{if false then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \Gamma, \Pi, e_3}$$

$$\textsc{Assertion-valid}\ \frac{\Gamma, \Pi \vdash t}{\Gamma, \Pi, \texttt{assert \{}\ t\ \texttt{\}} \rightsquigarrow \Gamma, \Pi, ()}$$

Figure 2: Small-step operational semantics: basic expressions

$$\textsc{op-context-left}\ \frac{\Gamma_1, \Pi_1, e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_2}{\Gamma_1, \Pi_1, e_1\ op\ e_3 \rightsquigarrow \Gamma_2, \Pi_2, e_2\ op\ e_3} \qquad \textsc{op-context-right}\ \frac{\Gamma_1, \Pi_1, e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_2}{\Gamma_1, \Pi_1, e_3\ op\ e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_3\ op\ e_2}$$

$$\textsc{Local-Variable-binding-context}\ \frac{\Gamma_1, \Pi_1, e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_2}{\Gamma_1, \Pi_1, \texttt{var}\ x = e_1\ \texttt{in}\ e_3 \rightsquigarrow \Gamma_2, \Pi_2, \texttt{var}\ x = e_2\ \texttt{in}\ e_3} \qquad \textsc{Variable-assignment-context}\ \frac{\Gamma_1, \Pi_1, e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_2}{\Gamma_1, \Pi_1, x \leftarrow e_1 \rightsquigarrow \Gamma_2, \Pi_2, x \leftarrow e_2}$$

$$\textsc{Conditional-context}\ \frac{\Gamma_1, \Pi_1, e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_1'}{\Gamma_1, \Pi_1, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \Gamma_2, \Pi_2, \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

Figure 3: Small-step operational semantics: context rules

$$\textsc{While-iterate}\ \frac{\Gamma, \Pi \vdash \phi_{inv}}{\begin{array}{c}\Gamma, \Pi, \texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \rightsquigarrow \\ \Gamma, \Pi, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done})\ \texttt{else}\ ()\end{array}}$$

Figure 4: Small-step operational semantics: loops

$$\textsc{Call}\quad \frac{\Gamma(f) = \mathsf{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \qquad \Gamma, \Pi_2 \vdash \phi_{pre}}{\Gamma, \Pi_1, (f\ z_1\ \cdots\ z_n) \ \rightsquigarrow\ \Gamma, \Pi_1, \mathsf{CallFrame}(\Pi_2, e_{body}, \phi_{post})}$$

$$\textsc{CallFrame-execution}\quad \frac{\Gamma_1, \Pi_1, e_1 \ \rightsquigarrow\ \Gamma_2, \Pi_2, e_2}{\Gamma_1, \Pi, \mathsf{CallFrame}(\Pi_1, e_1, \phi_{post}) \ \rightsquigarrow\ \Gamma_2, \Pi, \mathsf{CallFrame}(\Pi_2, e_2, \phi_{post})}$$

$$\textsc{Return}\quad \frac{\Gamma, \Pi_2[\text{result} \leftarrow v] \vdash \phi_{post}}{\Gamma, \Pi_1, \mathsf{CallFrame}(\Pi_2, v, \phi_{post}) \ \rightsquigarrow\ \Gamma, \Pi_1, v}$$

Figure 5: Small-step operational semantics: function calls

2.3.2 Execution of loops and function calls

Figure 4 presents the rule for executing loops. Notice how we incorporate in the semantics, as we did for assertions, the fact that the loop invariant must hold before evaluating the loop condition.

Figure 5 presents the rules for function calls. These rules make use of an additional pseudo-expression $\mathsf{CallFrame}(\Pi, e, Q)$ meaning that expression e is to be evaluated in the local environment Π (thus typically holding function parameters and local variables) and it must establish post-condition Q.

2.3.3 Explicitating Blocked Executions

We have now a complete set of rules for execution with assertion checking. As announced, the execution of an expression will be finite or infinite. If it is finite, it will terminate on a value, or get blocked when checking an annotation.

For the purpose of this paper, it is yet useful to make the blocked executions more explicit. In particular, we will need to distinguish the case of the `stuck` expression. We thus define an additional judgement of the form $\Gamma, \Pi, e \ \wr\ \xi$ where ξ is an outcome of a blocked execution, which is either Failure or Stuck. The former is associated with properties that do not hold and the latter is dedicated to the `stuck` expression. The rules for this judgement are given in Figure 6. These rules should be naturally completed by context rules, which are straightforward and omitted here.

It is important to notice that the rules for this blocking judgement are complementary to the rules of normal execution: for a given context and a given expression that is not a value, there is either a rule for normal execution that applies, or, exclusively, one rule for blocking execution that applies. That also explains why we omitted to include a rule for `stuck` before.

Definition 2.3 (Safety) *The execution of an expression e in a context Γ, Π is called* blocking *if*

$$\Gamma, \Pi, e \ \rightsquigarrow\ \Gamma_1, \Pi_1, e_1 \ \rightsquigarrow\ \Gamma_2, \Pi_2, e_2 \ \rightsquigarrow\ \cdots \Gamma_{n-1}, \Pi_{n-1}, e_{n-1} \ \wr\ \textit{Failure}$$

It is called safe *otherwise, that is either gets stuck:*

$$\Gamma, \Pi, e \ \rightsquigarrow\ \Gamma_1, \Pi_1, e_1 \ \rightsquigarrow\ \Gamma_2, \Pi_2, e_2 \ \rightsquigarrow\ \cdots \Gamma_{n-1}, \Pi_{n-1}, e_{n-1} \ \wr\ \textit{Stuck}$$

executes infinitely: there is an infinite sequence

$$\Gamma, \Pi, e \ \rightsquigarrow\ \Gamma_1, \Pi_1, e_1 \ \rightsquigarrow\ \Gamma_2, \Pi_2, e_2 \ \rightsquigarrow\ \cdots$$

$$\frac{\Gamma, \Pi \not\models \phi}{\Gamma, \Pi, \texttt{assert \{ } \phi \texttt{ \}} \not\rightarrow \mathsf{Failure}} \text{ ASSERTION-INVALID} \qquad \frac{}{\Gamma, \Pi, \texttt{stuck} \not\rightarrow \mathsf{Stuck}} \text{ STUCK}$$

$$\frac{\Gamma, \Pi \not\models \phi_{inv}}{\Gamma, \Pi, \texttt{while } c \texttt{ do invariant \{ } \phi_{inv} \texttt{ \} writes \{ } \vec{y} \texttt{ \} } e \texttt{ done} \not\rightarrow \mathsf{Failure}} \text{ WHILE-INVARIANT-FAILURE}$$

$$\frac{\Gamma(f) = \mathsf{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \qquad \Gamma, \Pi_2 \not\models \phi_{pre}}{\Gamma, \Pi_1, (f\ z_1\ \cdots\ z_n) \not\rightarrow \mathsf{Failure}} \text{ CALL-PRECONDITION-FAILURE}$$

$$\frac{\Gamma, \Pi_2[\mathsf{result} \leftarrow v] \not\models \phi_{post}}{\Gamma, \Pi_1, \mathsf{CallFrame}(\Pi_2, v, \phi_{post}) \not\rightarrow \mathsf{Failure}} \text{ RETURN-POSTCONDITION-FAILURE}$$

Figure 6: Small-step operational semantics: explicit blocking rules

or evaluates to a constant value in finite number of steps:

$$\Gamma, \Pi, e \rightsquigarrow \Gamma_1, \Pi_1, e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_2 \rightsquigarrow \cdots \Gamma_{n-1}, \Pi_{n-1}, e_{n-1} \rightsquigarrow \Gamma_n, \Pi_n, v$$

Definition 2.4 (Conformance) *An annotated program function* conforms *to its specifications if the corresponding execution is safe for a given set of values for its parameters and for global variables. Otherwise, if the execution blocks at some annotation, we say that it does not conform, or is non-conforming with this annotation.*

Example 2.5 (Examples 2.1 and 2.2 continued) *In both programs, the execution of* `main` *terminates in finite number of steps, reducing to the value* `()`*. So, these programs are conforming with their specifications.*

Example 2.6 *Consider the following program*

```
1   fun main () : unit
2     writes { x }
3   =
4     x ← 100;
5     while x >= 0 do
6      invariant { x >= 0 }
7      writes { x }
8      x ← x + 1
9    done;
10   assert { x < 0 }
```

The execution of `main()` *is infinite, within which the loop invariant always holds. The assertion is never reached. This program is thus conforming to its specifications.*

Example 2.7 *Consider the following program*

```
1   fun main () : unit
2     writes { x }
3   =
4     x ← 0;
5     while x < 100 do
6      invariant { x >= 0 }
7      writes { x }
8      if x > 50 then stuck;
9      x ← x + 1
10   done;
11   assert { x < 0 }
```

The execution of `main()` proceeds normally until it reaches the `stuck` statement. Since being stuck is considered safe, this program is conforming to its specifications.

Example 2.8 *Consider the following program*

```
1   fun main () : unit
2     writes { x }
3   =
4     x ← 0;
5     while x < 100 do
6      invariant { x <= 50 }
7      writes { x }
8      x ← x + 1
9    done;
10   assert { x < 0 }
```

The execution of `main()` proceeds normally until `x` gets value 51, when then the loop invariant is violated: this program is not conforming to its specifications. Imagine now that the loop invariant is replaced by `x <= 100`. In this case the execution proceeds to the end of the loop, when `x` gets value 100; then the assertion in line 10 fails and thus the program is not conforming to its specifications.

2.4 Generation of Verification Conditions

The goal of VC generation is to derive from a program a set of logical formulas whose validity implies the safety of the program. The ultimate goal of our work is to produce a suitable counterexample that helps the programmer understand what is wrong when the proof of the verification condition fails.

There are different variants of calculi to compute verification conditions. For the sake of this paper, we adopt the classical weakest-precondition calculus originally proposed by Dijkstra [10]. Other calculi should be convenient as well for the ultimate purpose of generating counterexample from proof failures.

The WP calculus is given through the function $\mathrm{WP}(e, Q)$. From an expression e and an expected post-condition Q, it returns another formula P that is called the weakest pre-condition of e with respect to Q. The formula P is the sufficient pre-condition on any initial context Γ, Π to guarantee the safe execution of e. The rules for computing WP on basic expressions are given in Figure 7.

Comparing with the standard WP calculus, we have to take into account the fact the we have expressions and not statements, and that a pseudo-variable for results may occur in Q. The rule for assignments could have been written directly as $Q[x \leftarrow t]$, yet we prefer the equivalent version with an extra variable because of the treatment that will come later on. The rule for assertions could have been written as $P \wedge Q$, yet our equivalent form is a good practice so as to use P as an extra hypothesis to prove Q. The rule for `stuck` captures the fact that executing `stuck` can never fail no matter the post-condition Q.

$$
\begin{array}{rcl}
\text{WP}(t, Q) & = & Q[\text{result} \leftarrow t] \\
\text{WP}(x \leftarrow t, Q) & = & \forall v.(v = t \rightarrow Q[x \leftarrow v]) \\
\text{WP}(\texttt{assert \{ } P \texttt{ \}}, Q) & = & P \wedge (P \rightarrow Q) \\
\text{WP}(\texttt{stuck}, Q) & = & \top \\
\text{WP}(\texttt{var } x \texttt{ = } e_1 \texttt{ in } e_2, Q) & = & \text{WP}(e_1, (\text{WP}(e_2, Q)[x \leftarrow \text{result}])) \\
\text{WP}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, Q) & = & \text{WP}(e_1, (\text{result} \rightarrow \text{WP}(e_2, Q)) \wedge (\neg\text{result} \rightarrow \text{WP}(e_3, Q)))
\end{array}
$$

where v is a fresh variable.

Figure 7: Rules for WP computation, basic expressions

$$
\begin{array}{l}
\text{WP}(\texttt{while } e_1 \texttt{ do invariant \{ } \phi_{inv} \texttt{ \} writes \{ } \vec{y} \texttt{ \} } e_2 \texttt{ done}, Q) = \\
\quad \phi_{inv} \ \wedge \forall \vec{v}.\ (\phi_{inv} \rightarrow \text{WP}(e_1, (\text{result} \rightarrow \text{WP}(e_2, \phi_{inv})) \wedge (\neg\text{result} \rightarrow Q)))[\vec{y} \leftarrow \vec{v}]
\end{array}
$$

$$
\text{WP}((f\ z_1\ \cdots\ z_n), Q) = \phi_{pre}[\vec{x} \leftarrow \vec{z}] \ \wedge \forall \vec{v}.\ (\phi_{post}[\vec{x} \leftarrow \vec{z}] \rightarrow Q)[\vec{y} \leftarrow \vec{v}]
$$

where $\vec{v}$ are fresh variables and f is declared as

$$
\begin{array}{l}
\texttt{fun } f\ (x_1 : \tau_1)\ \ldots\ (x_n : \tau_n)\ :\ \tau \\
\quad \texttt{requires \{ } \phi_{pre} \texttt{ \} ensures \{ } \phi_{post} \texttt{ \} writes \{ } \vec{y} \texttt{ \}}
\end{array}
$$

Figure 8: Rules for WP computation, loops and function calls

The rules for computing WP for loops and function calls are given in Figure 8. Here, the clauses for declaring written variables come to play: enumerated variables need to be universally quantified because the post-condition of function calls and the preservation of loop invariants must be shown valid independently of the values of these variables. In the rule for function calls, notice the importance of the typing condition requiring $\vec{y}$ and $\vec{z}$ to be disjoint – without it the rule would not be correct and indeed even the variable substitutions in that formula would be meaningless.

The set of verification conditions of a complete program is then the set of formulas

$$
\forall \vec{v}, \vec{w}, \text{result}.(\phi_{pre} \rightarrow \text{WP}(e_{body}, \phi_{post}))[\vec{x} \leftarrow \vec{v}, \vec{y} \leftarrow \vec{w}]
$$

for each function declared, where $\vec{v}$ and $\vec{w}$ are fresh variables corresponding respectively to the function parameters and the written global variables.

Proposition 2.9 (Partial correctness) *If the set of verification conditions of a program are valid formulas, then for any expression e, any formula Q, and any context Γ, Π, if $\Gamma, \Pi \vdash$ WP(e, Q) then e executes without blocking in Γ, Π, and if it ever terminates in some context Γ', Π' and value v, then $\Gamma', \{\textit{result} = v\} \cdot \Pi' \vdash Q$.*

Definition 2.10 (Subcontract Weakness) *An annotated program exposes a* subcontract weakness *if there is some annotation in the code that is not statically provable, although the program conforms to this annotation.*

Example 2.11 *Consider the following toy example.*

```
fun incr (n:int) : int
  writes { }
  ensures { result > n }
= n+1

fun f (x:int) : int
  writes { }
=
  var y = incr x in
  assert { y = x+1 };
  y
```

This program is safe: for any value for `x`*, the execution of* `f x` *is OK, including the assertion checking of the post-condition of* `incr` *and the* `assert` *statement.*

The VC for the function `incr` *is*

$$\forall n.\ \forall \mathit{result}.\ \mathit{result} = n + 1 \rightarrow \mathit{result} > n$$

which is a tautology, so the function `incr` *is safe.*

The VC for the function `f` *is*

$$\forall x.\ \forall \mathit{result}.\ \mathit{result} > x \rightarrow \forall y. y = \mathit{result} \rightarrow y = x + 1$$

where the premise of the implication comes from the post-condition of `incr`*. This formula is not a tautology and thus the program cannot proved safe despite being safe under every execution.*

This example exposes a subcontract weakness, namely in the post-condition of function `incr`*.*

Example 2.12 (Examples 2.1 and 2.2 continued) *As said in Example 2.5, these programs are conforming with their specifications. Yet, as notice in the introduction, the verification conditions are not provable: each of these programs exposes a subcontract weakness.*

Example 2.13 (Example 2.6 continued) *The VC corresponding to the final, in fact unreachable, assertion of this code is*

$$\forall x.\ x = 100 \rightarrow \forall x'.\ \neg(x' \geq 0) \wedge x' \geq 0 \rightarrow x' < 0$$

which is a valid formula. As surprising as it might be, this code is both safe and free of subcontract weaknesses.

Example 2.14 (Example 2.7 continued) *The VC corresponding to the final assertion of this code is*

$$\forall x.\ x = 0 \rightarrow \forall x'.\ \neg(x' < 100) \wedge x' \geq 0 \rightarrow x' < 0$$

which is a wrong statement. This safe code thus exposes a subcontract weakness. Indeed, a stronger loop invariant would be `x <= 50`*, which would allow to prove the final assertion, and would be preserved because of the* `stuck` *statement.*

Example 2.15 (a valid yet non-inductive loop invariant) *Consider the following example.*

```
fun main () =
  var x = 0 in
  var i = 0 in
  while i < 2 do
    invariant { i > 0 → x = 1 }
```

```
6       writes { i,x }
7       if i = 0 then x ← x + 1;
8       i ← i + 1
9     done
```

The concrete execution of the program is OK, including the run-time checking of the loop invariant. The VC for the preservation of the loop invariant, in the branch where $i = 0$, is

$$\forall i, x.\ (i > 0 \to x = 1) \wedge (i < 2) \to i = 0 \to (i + 1 > 0 \to x + 1 = 1)$$

This formula does not hold, for instance when $i = 0$ and $x = -1$. This illustrates a subcontract weakness for a loop invariant, not too weak to prove a subsequent assertion, but for proving that it is itself preserved by any loop iteration. This is an example of a loop invariant that is valid at run-time, but yet not provably preserved: it is a so-called non-inductive *loop invariant.*

2.5 Generating Candidate Counterexamples from Proof Failures

As roughly explained in the introduction, when a VC is passed to an SMT solver, its goal is negated and the solver is queried for satisfiability. If the solver answers unsat, we are done: the VC is valid logic statement. When the solver answers anything else, it may provide a candidate model of the input formula. We rely on a previous work by Dailler et al [8], which extensively studied how such a model can be turned into a candidate counterexample. Such a counterexample contains not only values for the parameters of the function under consideration, but also initial values for the global variables and values taken by mutable variables at different points in the program. The latter include the values for written variables in function calls and loops, that is specifically the variables involved in the `writes` clauses that we explicitly introduced in your μWhy language. More concretely, a counterexample is a collection of triples of the form (variable identifier, location in the source, value) that gives us values taken by the variables at the various location of the source code.

Example 2.16 *Consider again the code of toy Example 2.1:*

```
1  fun set_x (n:int) : unit
2    writes { x }
3    ensures { x > n }
4    =
5    x ← n + 1
6
7  fun main () : unit
8    writes { x }
9    =
10   x ← 0;
11   var temp :int = 2 in set_x temp;
12   assert { x = 3 }
```

The VC for the final assertion is

$$\forall x.x = 0 \to \forall x'.x' > 2 \to x' = 3$$

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 SMT solver is

$$x = 0 \wedge x' > 2 \wedge \neg(x' = 3)$$

to which the solver typically answers "sat" with the model $\{x = 0, x' = 4\}$*. The reconstruction of a candidate counterexample from such a model produces*

- *Variable* `x` *at line 10: 0*
- *Variable* `x` *at line 11: 4*

the first item corresponds to the value of `x` *after the assignment, whereas the second one corresponds to its value after the call to* `set_x`*.*

Example 2.17 *Consider again the code of toy Example 2.2:*

```
1  fun main () : unit
2    writes { x }
3    =
4    x ← 0;
5    while x < 10 do
6      invariant { x >= 0 }
7      writes { x }
8      x ← x + 1
9    done;
10   assert { x = 10 }
```

The VC for the final assertion is

$$\forall x.\, x = 0 \ \rightarrow \forall x'.\, x' \geq 0 \wedge x' \geq 10 \rightarrow x' = 10$$

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 SMT solver is*

$$x = 0 \wedge x' \geq 0 \wedge x' \geq 10 \wedge x' \neq 10,$$

to which the solver typically answers "sat" with the model $\{x = 0, x' = 11\}$*. The reconstruction of a candidate counterexample from such a model produces*

- *Variable* `x` *at line 4: 0*
- *Variable* `x` *at line 5: 11*

the first item corresponds to the value of `x` *after the first assignment, whereas the second one corresponds to its value at the loop exit.*

It is important to notice that because of the inherent incompleteness of SMT solvers (due to quantifiers or undecidable theories such as non-linear arithmetic), a candidate counterexample produced by this technique is never guaranteed to be valid. Our goal is thus to provide means to analyse such a candidate counterexample *a posteriori*, and categorise it as either a non-conformance, a subcontract weakness, or possibly an invalid counterexample.

CALL-PRECONDITION-FAILURE

$$\frac{\Gamma_1(f) = \text{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \leq i \leq n} \qquad \Gamma_1, \Pi_2 \not\vdash \phi_{pre}}{\Gamma_1, \Pi_1, (f\ z_1\ \cdots\ z_n) \ \wr_O\ \textsf{Failure}}$$

CALL-POSTCONDITION-STUCK

$$\frac{\begin{array}{c}\Gamma_1(f) = \text{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \leq i \leq n} \qquad \Gamma_1, \Pi_2 \vdash \phi_{pre} \\ \Gamma_2 = \Gamma_1[y_i \leftarrow O(p, y_i)]_{1 \leq i \leq m} \qquad v = O(p, \text{result}) \qquad \Gamma_2, \{\text{result} \leftarrow v\} \cdot \Pi_2 \not\vdash \phi_{post}\end{array}}{\Gamma_1, \Pi_1, [p](f\ z_1\ \ldots\ z_n) \ \wr_O\ \textsf{Stuck}}$$

CALL-SUCCESS

$$\frac{\begin{array}{c}\Gamma_1(f) = \text{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \leq i \leq n} \qquad \Gamma_1, \Pi_2 \vdash \phi_{pre} \\ \Gamma_2 = \Gamma_1[y_i \leftarrow O(p, y_i)]_{1 \leq i \leq m} \qquad v = O(p, \text{result}) \qquad \Gamma_2, \{\text{result} \leftarrow v\} \cdot \Pi_2 \vdash \phi_{post}\end{array}}{\Gamma_1, \Pi_1, [p](f\ z_1\ \ldots\ z_n) \overset{O}{\rightsquigarrow} \Gamma_2, \Pi_1, v}$$

Figure 9: Giant-step operational semantics: Function calls

3 Giant-Step Assertion-Checking

We propose an original method for categorising counterexamples, that is the *giant step* run-time assertion checking. In this section we first formally define this semantics, and then we explain how it can be used to categorise counterexamples.

3.1 Giant-Step Assertion-Checking Semantics

The giant-step execution differs from the standard execution only for function calls and while loops. Instead of evaluating the function body and iterating over the loop body, an *oracle* is used to retrieve the return value of a function call and the values of the variables written by the function and loop. In other words, the execution of a function or a loop is done in one single step, thus the term "giant-step". The intention is to use a candidate counterexample as the oracle, as it will be explained in Section 3.2.

The semantic judgement for giant-step semantics is denoted as $\Gamma_1, \Pi_1, e_1 \overset{O}{\rightsquigarrow} \Gamma_2, \Pi_2, e2$, meaning that in context (Γ_1, Π_1), expression e_1 reduces in one giant step, under oracle O, to context (Γ_2, Π_2) and expression e_2. Sometimes we will annotate expressions under the form $[p]e$ where p represents some lexical position of expression e, that identifies it uniquely, and may be used in the oracle. The oracle is thus a mapping $O : \mathit{Pos} \times \mathit{Ident} \to \mathit{Value}$. It is worth to notice that a mapping from positions is sufficient because, as it will be seen below, every position is evaluated at most once under the giant-step semantics.

The rules for executing basic expressions are indeed identical (modulo the notation of judgement) to those of small-step semantics. We thus do not repeat them. The rules for function calls are given in Figure 9. The three rules can be seen as three alternatives. The first rule CALL-PRECONDITION-FAILURE checks for the validity of the pre-condition in the current context and is similar to the rule of the same name in Figure 6, that is it blocks if that pre-condition is not valid. On the contrary, the second rule CALL-POSTCONDITION-STUCK applies when the pre-condition is valid, and performs a giant step: it queries the oracle for final values of the variables potentially modified by the call, and also queries for a return value. With those values from the oracle, it checks the post-condition. If that post-condition is invalid, it means that the oracle is bad, in the sense that it does not provide values that are possible for a valid function call. This is the reason why the rule concludes to Stuck. As a third alternative, the third rule CALL-SUCCESS applies when the post-condition is instead valid, and thus allows to continue the

$$\textsc{While-invariant-initialisation-failure}\quad \frac{\Gamma, \Pi \not\vdash \phi_{inv}}{\Gamma, \Pi, \texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \not\rightsquigarrow_O \textsf{Failure}}$$

$$\textsc{While-any-iteration-stuck}\quad \frac{\Gamma_1, \Pi_1 \vdash \phi_{inv} \qquad (\Gamma_2, \Pi_2) = (\Gamma_1, \Pi_1)[y_i \leftarrow O(p, y_i)]_{1 \le i \le k} \qquad \Gamma_2, \Pi_2 \not\vdash \phi_{inv}}{\Gamma_1, \Pi_1, [p]\texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \not\rightsquigarrow_O \textsf{Stuck}}$$

$$\textsc{While-any-iteration}\quad \frac{\Gamma_1, \Pi_1 \vdash \phi_{inv} \qquad (\Gamma_2, \Pi_2) = (\Gamma_1, \Pi_1)[y_i \leftarrow O(p, y_i)]_{1 \le i \le k} \qquad \Gamma_2, \Pi_2 \vdash \phi_{inv}}{\begin{array}{c}\Gamma_1, \Pi_1, [p]\texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \overset{O}{\rightsquigarrow} \\ \Gamma_2, \Pi_2, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{assert \{}\ \phi_{inv}\ \texttt{\}}; \texttt{stuck})\ \texttt{else}\ ()\end{array}}$$

Figure 10: Giant-step operational semantics: loops

execution with the return value got from the oracle.

Example 3.1 (Toy Example 2.1 continued) *Consider again the code of Example 2.1:*

```
1  fun set_x (n:int) : unit
2    writes { x }
3    ensures { x > n }
4    =
5    x ← n + 1
6
7  fun main () : unit
8    writes { x }
9    =
10   x ← 0;
11   var temp :int = 2 in set_x temp;
12   assert { x = 3 }
```

As seen in Example 2.16, a candidate counterexample is generated from the SMT solver model:

- *Variable* `x` *at line 10: 0*
- *Variable* `x` *at line 11: 4*

This counterexample can be used as an oracle to execute `main()` *with giant steps. The call to* `set_x` *queries the oracle, providing the value 4 for the new value of* `x`*, which perfectly satisfies the post-condition. Later on, the execution of the assertion fails. Since concrete execution was successful, we can suspect a subcontract weakness, that is, an insufficient post-condition for* `set_x`*.*

Similar ideas can be followed for the giant-step execution of while loops. The rules for executing loops are given in Figure 10. The rule WHILE-INVARIANT-INITIALISATION-FAILURE is similar to the rule WHILE-INVARIANT-FAILURE of Figure 6: it checks if the loop invariant is true when entering the loop and fails if not. Otherwise, if the loop invariant is initially valid, then the two other rules perform a "giant step" to an *arbitrary iteration* of the loop, where the values of the modified variables at the beginning of that iteration are taken from the oracle. The rule WHILE-ANY-ITERATION-STUCK then

checks for the validity of the loop invariant when entering this iteration. If the loop invariant does not hold, it means that the oracle provides unsuitable values for the written variables, and thus we get stuck. Otherwise, as the rule WHILE-ANY-ITERATION states, if the condition of the loop holds, its body is executed, followed by a check of the loop invariant, under the form of an assertion. It means that if the loop invariant is not re-established, then we identified a failure. On the contrary, if the invariant is re-established then the oracle does not expose a good execution and we are stuck. The idea here is that we don't want to continue the execution after executing an arbitrary iteration; we also don't want to execute the body of the loop multiple times. So the only way we can get through the loop in the giant-step execution is if the oracle provides us with values that satisfy the invariant and make the loop condition false as in rule WHILE-ANY-ITERATION.

Example 3.2 (Toy Example 2.2 continued) *Consider again the code of Example 2.2:*

```
1  fun main () : unit
2    writes { x }
3    =
4    x ← 0;
5    while x < 10 do
6      invariant { x >= 0 }
7      writes { x }
8      x ← x + 1
9    done;
10   assert { x = 10 }
```

As seen in Example 2.17, a candidate counterexample is generated from the SMT solver model:

- *Variable x at line 4: 0*
- *Variable x at line 5: 11*

This counterexample can be used as an oracle to execute `main()` *with giant steps. When reaching the loop at line 5, giant-step execution first checks that the loop invariant is initially true, which is the case. Then a new value for* `x` *is queried from the oracle, providing the value 11. For this value the loop invariant is valid and the rule* WHILE-ANY-ITERATION *applies. The execution proceeds by evaluating the loop condition, which is false, and then proceeds with what comes after the loop, that is the assertion, which fails. Since concrete execution was successful, we can again suspect a subcontract weakness, that is, an insufficient loop invariant.*

Example 3.3 *Consider again the code of Example 2.15:*

```
1  fun main () =
2    var x = 0 in
3    var i = 0 in
4    while i < 2 do
5      invariant { i > 0 → x = 1 }
6      writes { i,x }
7      if i = 0 then x ← x + 1;
8      i ← i + 1
9    done
```

As seen in Example 2.15, the preservation of the loop invariant could not be proved, a candidate counterexample being generated from the SMT solver model:

	Giant-step RAC			
Small-step RAC	Failure	Normal	Stuck	Incomplete
Failure matches goal	Non-conformity			
Failure elsewhere	Bad CE (invalid assertion somewhere else)			
Stuck	Invalid Assumption			
Normal	Sub-contract weakness	Bad CE	Bad CE	Incomplete
Incomplete	Non-conformity or sub-contract weakness	Incomplete	Bad CE	Incomplete

Figure 11: Categorisation of candidate counterexamples, as a function of the results from the RAC using small-step semantics and giant-step semantics

- *Variable* `i` *at line 4: 0*
- *Variable* `x` *at line 4: -1*

This counterexample can be used as an oracle to execute `main()` *with giant steps. When reaching the loop at line 4, the values for* `i` *and* `x` *are queried from the oracle. For these value the loop invariant is valid and the loop condition is true. Giant-step execution proceeds by execution the loop body, leading to values 1 for* `i` *and 0 for* `x`*. The loop invariant is checked again, and fails. We must suspect a subcontract weakness, that is a too weak loop invariant.*

3.2 Automatic Classification of Counterexamples

A *candidate* counterexample may witness a non-conformity in a program annotation, a subcontract weakness, or it may in fact *not* be a counterexample at all, if it does not lead to any violation or if it provides values that contradict the assumptions.

A candidate counterexample is classified by executing a call to the function that contains the VC using both concrete and giant-step executions, and by comparing the results of the two executions. For both executions, the values from the candidate counterexample are used to initialise global variables and parameters of the target function. For the giant-step execution, the values from the counterexample are also used as the oracle.

Figure 11 summarises the classification based on the results of the two executions. When the concrete execution fails the counterexample comprises a witness of a non-conformity between program and the assertion. When the concrete execution terminates normally, but the giant-step execution fails due to the violation of an assertion, the counterexample shows that whereas the contracts are correct during the concrete execution, some postconditions or loop invariants permit values that result in violation of latter contracts. In this case, the counterexample witnesses a global subcontract-weakness. When the concrete execution and giant-step execution terminate normally, *i.e.* no contract has been violated during either execution, the candidate counterexample is bad. When the giant-step execution gets stuck, it signals that the counterexample contains values that violate the assumptions of the execution, and the candidate counterexample is likewise considered bad.

A final remark about Figure 11: the last line and the last column concern the case the result of RAC is *incomplete*. Indeed, in the semantic rules, the decidability of the formulas was assumed. However, in practice, the validity of some formulas may not be decidable during RAC execution: we will detail this issue further in Section 4.4. It is thus in these cases that the execution result is declared incomplete. In our table, when the result of the concrete execution is incomplete and an assertion is violated during giant-step execution, the counterexample witnesses either a non-conformity between the program and the assertion or a global sub-contract weakness, but we cannot decide which one of them.

4 Extensions and Implementation

4.1 Refering to the past with labels

In practice, it is frequent that one wants to refer to the past in the specifications. The most common case is the use of a `old` construct in post-conditions, to refer to values of variables when entering the corresponding program function.

Example 4.1 *The following example shows a function* `incr` *with a post-condition stating that the new value of* `x` *is greater than its previous value.*

```
1  fun incr_x () : unit
2    writes { x }
3    ensures { x > old x }
4    =
5    x ← x + 1
```

A more general case is the use of a `at` construct to refer to a value at a given program label.

Example 4.2 *The following example illustrates a loop invariant and an assertion refering to past values of* `x`*.*

```
1  fun main () : unit
2    writes { x }
3    =
4    x ← x + 1;
5    label L in
6    x ← x + x;
7    assert { x > x at L };
8    var i = 0 in
9    while i < 10 do
10     invariant { x > x at L + i }
11     x ← x + 1;
12     i ← i + 1;
13   done;
14   assert { x > x at L + 10 }
```

Those label and `old` and `at` constructs are supported by the generation of candidate counterexample [8]. In order to categorise those candidate counterexamples we need to support them in both the small-step and the giant-step execution of programs.

Technically, supporting these constructs is not particularly difficult: we just have to generalise the contexts of execution. Instead of being maps from variables to values, the contexts Γ and Π should also be parameterised by labels, that is being maps from labels and variables to values. To make such a presentation uniform, we introduce a default label `Here` which denotes the current state. The rules presented in Figure 12 are the important changed rules and new rules in the small-step semantics. Notice how the traversal of a label declaration performs a kind of snapshot of the current context. The rules for giant-step semantics are extended following the same principles.

The rules for computing WP must be extended accordingly as follows:

$$\begin{array}{rcl} \mathrm{WP}(x \leftarrow t, Q) & = & \forall v.(v = t \rightarrow Q[(x\ \texttt{at}\ Here) \leftarrow v]) \\ \mathrm{WP}(\texttt{label}\ L\ \texttt{in}\ e, Q) & = & \mathrm{WP}(e, Q)[(x\ \texttt{at}\ L) \leftarrow (x\ \texttt{at}\ Here)]_{x \text{ any variable}} \end{array}$$

GLOBAL VARIABLE

$$\frac{\Gamma(Here)(x) = v}{\Gamma, \Pi, x \rightsquigarrow \Gamma, \Pi, v}$$

GLOBAL-VARIABLE-ASSIGNMENT

$$\frac{x \in dom(\Gamma(Here))}{\Gamma, \Pi, x \leftarrow v \rightsquigarrow \Gamma[Here, x \leftarrow v], \Pi, ()}$$

LABEL DECLARATION

$$\frac{}{\Gamma, \Pi, \texttt{label}\ L\ \texttt{in}\ e \rightsquigarrow \Gamma[L \leftarrow \Gamma(Here)], \Pi[L \leftarrow \Pi(Here)], e}$$

Figure 12: Operational semantics for labels

WHILE-ITERATE

$$\frac{\Gamma, \Pi \vdash \phi_{inv}}{\Gamma, \Pi, \texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} variant \{}\ t_{var}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \rightsquigarrow \Gamma, \Pi, \left(\begin{array}{l}\texttt{label}\ L\ \texttt{in} \\ \texttt{if}\ c\ \texttt{then}\ (e; \texttt{assert \{}\ t_{var} < t_{var}\ \texttt{at}\ L\ \texttt{\};} \\ \qquad\qquad \texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} variant \{}\ t_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done}) \\ \quad \texttt{else}\ ()\end{array}\right)}$$

RECURSIVE-CALL

$$\frac{\Gamma(f) = \text{Func}(\vec{x}, \phi_{pre}, t_{var}, \phi_{post}, \vec{y}, e_{body}) \quad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \quad \Gamma, \Pi_2 \vdash \phi_{pre} \quad \Gamma, \Pi_2 \vdash t_{var} < t_{var}\ \texttt{at}\ Init_f}{\Gamma, \Pi_1, (f\ z_1\ \cdots\ z_n) \rightsquigarrow \Gamma, \Pi_1, \mathsf{CallFrame}(\Pi_2, \texttt{label}\ Init_f\ \texttt{in}\ e_{body}, \phi_{post})}$$

Figure 13: Small-step operational semantics: variant clauses

and the VC for a whole program function is

$$\forall \vec{v}, \vec{w}, \text{result}.(\phi_{pre} \rightarrow \text{WP}(e_{body}, \phi_{post}))[\vec{x} \leftarrow \vec{v}, \vec{y} \leftarrow \vec{w}, \texttt{old}\ \vec{y} \leftarrow \vec{w}]$$

4.2 Verification of Termination

Another useful extension is the support for *total correctness*, that is the ability to also prove termination of programs. This concerns both the termination of loops and the termination of recursive or mutually recursive function calls. The classical approach to achieve verification of termination is to add specific annotation clauses, namely `variant` clauses.

The syntax of our μWhy language must thus be extended to support such clauses, as follows.

$$\begin{array}{rcl} d & ::= & \texttt{fun}\ f\ (x_1 : \tau_1)\ \ldots\ (x_n : \tau_n)\ :\ \tau \\ & & \texttt{requires \{}\ \phi_{pre}\ \texttt{\} variant \{}\ t_{var}\ \texttt{\} ensures \{}\ \phi_{post}\ \texttt{\} writes \{}\ y_1, \ldots, y_k\ \texttt{\} =}\ e \\ e & ::= & \texttt{while}\ e_1\ \texttt{do invariant \{}\ \phi_{\text{inv}}\ \texttt{\} variant \{}\ t_{var}\ \texttt{\} writes \{}\ y_1, \ldots, y_k\ \texttt{\}}\ e_2\ \texttt{done} \end{array}$$

For simplicity we assume here that such variants are integer expressions, that must be strictly decreasing while remaining non-negative. The extension to an arbitrary datatype and well-founded ordering relation is straightforward. We also restrict ourselves to direct recursion: extension to mutual recursion is slightly more involved technically but has no intrinsic additional difficulty.

$$\text{WP}(\texttt{while}\ e_1\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}}\ \texttt{variant \{}\ t_{var}\ \texttt{\}}\ \texttt{writes \{}\ \vec{y}\ \texttt{\}}\ e_2\ \texttt{done}, Q) =$$
$$\phi_{inv}\ \wedge \forall \vec{v}.\ (\phi_{inv} \rightarrow \text{WP}(\texttt{label}\ L\ \texttt{in}\ e_1,$$
$$(\text{result} \rightarrow \text{WP}(e_2, \phi_{inv} \wedge t_{var} < t_{var}\ \texttt{at}\ L)) \wedge (\neg\text{result} \rightarrow Q)))[\vec{y} \leftarrow \vec{v}]$$

$$\text{WP}((f\ z_1\ \cdots\ z_n), Q) = \phi_{pre}[\vec{x} \leftarrow \vec{z}] \wedge t_{var} < t_{var}\ \texttt{at}\ Init_f \wedge \forall \vec{v}.\ (\phi_{post}[\vec{x} \leftarrow \vec{z}] \rightarrow Q)[\vec{y} \leftarrow \vec{v}]$$

where $\vec{v}$ are fresh variables and f is declared as

$$\texttt{fun}\ f\ (x_1 : \tau_1)\ \ldots\ (x_n : \tau_n)\ :\ \tau$$
$$\texttt{requires \{}\ \phi_{pre}\ \texttt{\}}\ \texttt{variant \{}\ t_{var}\ \texttt{\}}\ \texttt{ensures \{}\ \phi_{post}\ \texttt{\}}\ \texttt{writes \{}\ \vec{y}\ \texttt{\}}$$

Figure 14: Rules for WP computation, loops and function calls with variants

CALL-VARIANT-FAILURE

$$\frac{\Gamma_1(f) = \text{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \qquad \Gamma_1, \Pi_2 \vdash \phi_{pre} \qquad \Gamma, \Pi_2 \nvdash t_{var} < t_{var}\ \texttt{at}\ Init_f}{\Gamma_1, \Pi_1, (f\ z_1\ \cdots\ z_n)\ \not\leadsto_O\ \textsf{Failure}}$$

WHILE-ANY-ITERATION-WITH-VARIANT

$$\frac{\Gamma_1, \Pi \vdash \phi_{inv} \qquad \Gamma_2 = \Gamma_1[y_i \leftarrow O(p, y_i)]_{1 \le i \le k} \qquad \Gamma_2, \Pi \vdash \phi_{inv}}{\begin{array}{c}\Gamma_1, \Pi, [p]\texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}}\ \texttt{writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \overset{O}{\rightsquigarrow} \\ \Gamma_2, \Pi, \texttt{label}\ L\ \texttt{in if}\ c\ \texttt{then}\ (e; \texttt{assert \{}\ t_{var} < t_{var}\ \texttt{at}\ L\ \texttt{\}}; \texttt{assert \{}\ \phi_{inv}\ \texttt{\}}; \texttt{stuck})\ \texttt{else}\ ()\end{array}}$$

Figure 15: Rules for giant-step execution, checking of variants

Figure 13 displays the rules for small-step execution of loops and function calls, extended to support run-time checking of variant clauses. We make use of the labels introduced in the previous section.

The rules for computing WP can be extended accordingly, as shown in Figure 14, where the VC for a whole program function `f` is completed by the declaration of the $Init_f$ label:

$$\forall \vec{v}, \vec{w}, \text{result}.(\phi_{pre} \rightarrow \text{WP}(\texttt{label}\ Init_f\ \texttt{in}\ e_{body}, \phi_{post}))[\vec{x} \leftarrow \vec{v}, \vec{y} \leftarrow \vec{w}]$$

Finally, we have to update the giant-step semantics for checking variants. Figure 15 shows the two rules that must be added.

4.3 Dealing with "for" loops

Let us now consider the inclusion of for-loops. For simplification purposes we assume that the lower are upper bounds of the loop are values (generalising them to expressions does not impose any additional complication). The syntax of μWhy is extended as follows:

$$e \quad ::= \quad \texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}}\ \texttt{writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done}$$

The small-step operational semantics is given in Figure 16. As shown in the first rule, the particularity of for-loops is that the invariant is not even taken into account when $v_1 > v_2 + 1$ (v_1 and v_2 stand

FOR-LOOP-EMPTY-RANGE

$$\frac{v_1 > v_2 + 1}{\Gamma, \Pi, \texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \rightsquigarrow \Gamma, \Pi, ()}$$

FOR-LOOP-INVARIANT-FAILURE

$$\frac{v_1 \leq v_2 + 1 \qquad \Gamma, (i, v_1) \cdot \Pi \nvdash \phi_{inv}}{\Gamma, \Pi, \texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \lightning \textsf{Failure}}$$

FOR-LOOP-ITERATE

$$\frac{v_1 \leq v_2 + 1 \qquad \Gamma, (i, v_1) \cdot \Pi \vdash \phi_{inv}}{\begin{array}{c}\Gamma, \Pi, \texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \rightsquigarrow \\ \Gamma, \Pi, (\texttt{var}\ i = v_1\ \texttt{in}\ e)\ \texttt{;}\ \texttt{for}\ i\ =\ v_1 + 1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done}\end{array}}$$

Figure 16: Small-step operational semantics: for-loops

$$\begin{array}{l}\mathrm{WP}(\texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done}, Q) = \\ \quad (v_1 \leq v_2 + 1 \to (\phi_{inv}[i \leftarrow v_1] \wedge \\ \quad\quad \forall \vec{v}, i.\ (\phi_{inv} \to (v_1 \leq i \leq v_2 \to \mathrm{WP}(e, \phi_{inv}[i \leftarrow i + 1])) \wedge (i = v_2 + 1 \to Q))[\vec{y} \leftarrow \vec{v}])) \wedge \\ \quad (v_1 > v_2 + 1 \to Q)\end{array}$$

Figure 17: Rules for WP computation of for-loops

respectively for the lower and upper bound). The rational behind this is that the variable i is bound locally, might occur in the invariant, and its value should range between v_1 and $v_2 + 1$: when $v_1 > v_2 + 1$ it does not make sense to instantiate i in the invariant. The WP calculus shown in Figure. 17 takes this into account and so does the giant-step semantics in Figure. 18. The second and third rule in Figure. 16 do not bring any additional complexity: note only that the variable i is bind locally and we increment the lower bound when expanding the loop in the third rule.

Let us now focus on the giant-step semantics, given by rules of Figure. 18. The first rule is the same as the one for small steps. The third rule queries the oracles for the assigned variables and also for i: if the value of i does not lie between v_1 and $v_2 + 1$ the execution gets stuck. The fourth rule also gets stuck if the invariant does not hold with the retrieved values from the oracles similarly to the corresponding rule for while-loops. Finally, the last rule expands the for-loop. Note that the value of i must lie between v_1 and $v_2 + 1$. We use a conditional statement to decide whether the body of the loop should be executed and then the invariant checked (getting stuck if the invariant holds) or simply terminate the execution with the value $()$.

Example 4.3 *Let us consider the example below*

```
1 fun foo () : int
2   ensures { result = 2 }
3   = var x : int = 0 in
4     for i = 1 to 2 do
5       invariant { x >= 0 } writes {x}
6       x ← x + 1
```

FOR-LOOP-EMPTY-RANGE

$$\frac{v_1 > v_2 + 1}{\Gamma, \Pi, \texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \overset{O}{\rightsquigarrow} \Gamma, \Pi, ()}$$

FOR-LOOP-INVARIANT-INITIALISATION-FAILURE

$$\frac{v_1 \leq v_2 + 1 \qquad \Gamma, (i, v_1) \cdot \Pi \nvdash \phi_{inv}}{\Gamma, \Pi, \texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \not\rightsquigarrow_O \textsf{Failure}}$$

FOR-LOOP-BAD-INDEX

$$\frac{v_1 \leq v_2 + 1 \qquad \Gamma, (i, v_1) \cdot \Pi \vdash \phi_{inv} \qquad \neg(v_1 \leq O(p, i) \leq v_2 + 1)}{\Gamma, \Pi, [p]\texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \not\rightsquigarrow_O \textsf{Stuck}}$$

FOR-LOOP-INVARIANT-STUCK

$$\frac{\begin{array}{c} v_1 \leq v_2 + 1 \qquad \Gamma, (i, v_1) \cdot \Pi \vdash \phi_{inv} \\ v_1 \leq O(p, i) \leq v_2 + 1 \qquad (\Gamma_1, \Pi_1) = (\Gamma, \Pi)[y_j \leftarrow O(p, y_j)]_j \qquad \Gamma_1, (i, O(p, i)) \cdot \Pi_1 \nvdash \phi_{inv} \end{array}}{\Gamma, \Pi, [p]\texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \not\rightsquigarrow_O \textsf{Stuck}}$$

FOR-LOOP-INVARIANT-PRESERVATION

$$\frac{\begin{array}{c} v_1 \leq v_2 + 1 \qquad \Gamma, (i, v_1) \cdot \Pi \vdash \phi_{inv} \\ v_1 \leq O(p, i) \leq v_2 \qquad (\Gamma_1, \Pi_1) = (\Gamma, \Pi)[y_j \leftarrow O(p, y_j)]_j \qquad \Gamma_1, (i, O(p, i)) \cdot \Pi_1 \vdash \phi_{inv} \end{array}}{\begin{array}{c} \Gamma, \Pi, \texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \overset{O}{\rightsquigarrow} \\ \Gamma_1, (i, O(p, i)) \cdot \Pi_1, (e\ \texttt{;}\ i \leftarrow i + 1\ \texttt{; assert \{}\ \phi_{inv}\ \texttt{\} ; stuck}) \end{array}}$$

FOR-LOOP-EXIT

$$\frac{\begin{array}{c} v_1 \leq v_2 + 1 \qquad \Gamma, (i, v_1) \cdot \Pi \vdash \phi_{inv} \\ O(p, i) = v_2 + 1 \qquad (\Gamma_1, \Pi_1) = (\Gamma, \Pi)[y_j \leftarrow O(p, y_j)]_j \qquad \Gamma_1, (i, O(p, i)) \cdot \Pi_1 \vdash \phi_{inv} \end{array}}{\Gamma, \Pi, \texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \overset{O}{\rightsquigarrow} \qquad \Gamma_1, \Pi_1, ()}$$

Figure 18: Giant-step operational semantics: for-loops

```
7       done; x
```

The VC for the post-condition cannot be proved, an SMT solver will possibly return a model which will be turned into the candidate counterexample

- x *at line 3 : 0*
- i *at line 4 : 3*
- x *at line 4 : 0*

The typical run-time assertion checking will terminate normally. The giant-step execution of the "for" loop will check the invariant holds for both value before and after the loop for x (0 in both cases), then jump over the loop since $i = 3$, and then fail on the check of the post-condition. The CE is thus categorised as a sub-contract weakness. Indeed a suitable stronger loop invariant should be `i=x+1`.

Notice that the rule for computing the WP of a for loop could be stated in a slight different, though equivalent, form:

$$
\begin{array}{l}
\mathrm{WP}(\texttt{for}\ i\ =\ v_1\ \texttt{to}\ v_2\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\}\ writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done}, Q) = \\
\quad (v_1 \leq v_2 + 1 \rightarrow (\\
\qquad \phi_{inv}[i \leftarrow v_1] \wedge \\
\qquad \forall \vec{v}, i.\ (\phi_{inv} \rightarrow (v_1 \leq i \leq v_2 \rightarrow \mathrm{WP}(e, \phi_{inv}[i \leftarrow i + 1]))) \wedge \\
\qquad (\phi_{inv}[i \leftarrow v_2 + 1] \rightarrow Q)[\vec{y} \leftarrow \vec{v}])) \wedge \\
\quad (v_1 > v_2 + 1 \rightarrow Q)
\end{array}
$$

With such a formula, a VC coming after the loop would not contain the loop index as a free variable, meaning that an SMT solver model would not contain any value for that index. Accordingly, the rule FOR-LOOP-EXIT should not have $O(p, i) = v_2 + 1$ as a premise, but instead something like $O(p, i)$ is undefined.

Example 4.4 (Example 4.3 continued) *With the variant of the WP computation proposed above, the candidate CE*

- x *at line 3 : 0*
- x *at line 4 : 0*

would be proposed, without any value for i. That value being absent, the giant-step should proceed as it was the last index plus 1, and thus jump over the loop, leading to the same categorisation as before.

4.4 Why3 Implementation of Assertion Checking

In Why3, the annotation language is not executable [12]. It means in practice that checking the validity of an assertion is a nontrivial task. Our implementation does its best to decide whether an annotation is valid or not, using a combination of techniques.

When a program annotation is encountered during RAC execution, a Why3 proof task is generated from imported definitions and the values in the execution context. The validity of the task is then checked in a procedure with three steps that are tried sequentially until one is able to decide the validity of the task, i.e. is not *incomplete* (Figure 19). First, Why3's term reduction engine is applied to compute the validity of the task using the transformation `compute_in_goal`. If the task goal can be reduced to *true* (or *false*), the assertion is valid (or invalid). Otherwise the reduction engine is incomplete, and the task is dispatched to a configurable, external prover. If the external prover can show the validity (or invalidity) of the task, the assertion is valid (or invalid) in the current environment.

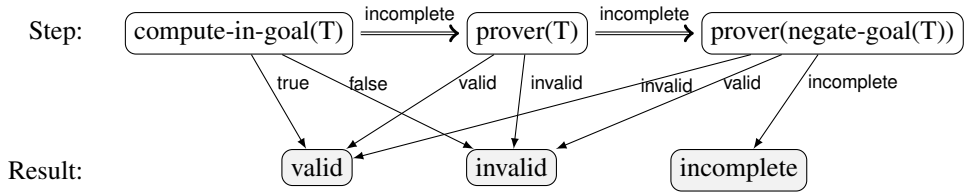


Figure 19: Procedure for checking assertions during execution

Generally these two steps cannot identify invalid assertions when this requires the instantiation of quantified preconditions: the reduction engine on the one hand does not instantiate quantified preconditions at all, and provers on the other hand are typically unable to identify invalid assertions. To resolve this limitation, the external prover is applied on the Why3 task with *negated goal*, and the assertion is considered invalid if the prover shows the validity of the task with negated goal. (This step is sound since the law of excluded middle is assumed in the logic of Why3).

If the assertion is found to be valid by one of the three steps, the RAC execution continues. Valid assertions from the syntactic scope in the program are retained as additional hypotheses for subsequent assertion checks. If the assertion is found to be invalid, the RAC execution fails. And if the procedure is incomplete, i.e., the validity of the task could not be decided, the RAC execution always terminates as *incomplete* during the classification of proof failures.

Reduction of bounded universal quantifications Since the annotation language is not executable, the reduction engine has previously not attempted to reduce any universal quantified terms. To extend its applicability, we added basic support for the reduction of *bounded* universal quantifications of the form `forall` i`: int.` $l\ op_l\ i\ op_u\ u \rightarrow t$, where l and u are the lower and upper bounds of the quantifications, and op_l and op_u may be the strict inequality $<$ or the non-strict inequality $\leq$. Let l' and u' be the *inclusive* bounds of the quantifications:

$$l' := \begin{cases} l+1 & \text{if } op_l \text{ is strict} \\ l & \text{otherwise} \end{cases} \qquad u' := \begin{cases} u-1 & \text{if } op_u \text{ is strict} \\ u & \text{otherwise} \end{cases}$$

The reduction applies if the set of values involved in the quantification is bound by a given limit B, *i.e.*, if $u' - l' \leq B$. The bound B defaults to 10 in the current implementation. In this case the formula is reduced to $\mathtt{t}[i \leftarrow l'] \wedge \ldots \wedge \mathtt{t}[i \leftarrow u']$.

4.5 Why3 Implementation of CE generation and Categorisation

The pipeline for generating counterexamples in Why3 is shown in Figure 20. First, Why3 generates the verification conditions (VC) for the program usually using its own implementation of a WP calculus. Transformations are used to simplify the VC into one or more verification goals, and to add information to proof task to help the reconstruction of counterexamples later. The categorisation of the proof failure using RAC execution requires a correspondence between syntactical annotations and verification goals, which can be obtained by splitting the VC using a transformation. For each verification goal, the SMT input contains the preconditions and the negation of the verification goal.

In practice, not only one but three models are requested from the SMT solver with increasing availability of hypotheses: (1) with no hypothesis, (2) without quantified hypotheses and (3) with all hypothe-

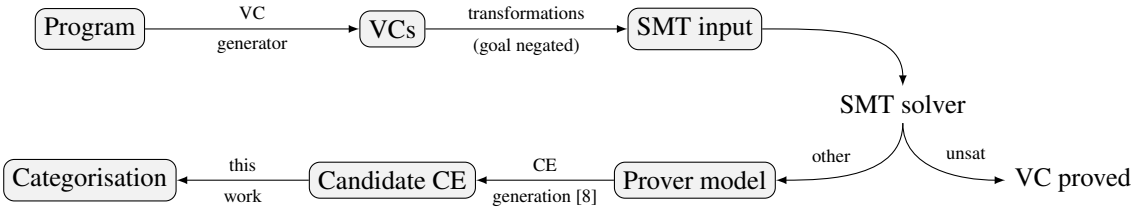


Figure 20: Pipeline of the Why3 counterexample (CE) generation and categorisation

ses. Candidate counterexamples, which map variables at specific program locations to values, are then reconstructed from the solver models (see Section 2.5).

The categorisation of proof failures as described in this report is based on values from the candidate counterexamples and comprises the following steps. First, the program function is identified to which the verification goal corresponds. (Specifically, the approach is only applicable to verification goals that originate in program definitions, and not to verification goals coming from logical definitions such as a goals, lemmas, or type invariants.) The function is then executed twice using Why3's interpreter with run-time assertion checking (RAC): using small-step semantics and using giant-step semantics. Initial values of global variables, and the values of the function parameters are selected from the candidate counterexample for both executions. Result values of functions calls and the values of variables written by functions and loops are taken from the candidate counterexample during giant-step execution. Each RAC execution may terminate normally, fail at some assertion, get stuck due to a failed assumption, or be *incomplete*. Incompleteness is a result that can show up in practice for various reasons, including missing values from the candidate counterexample, missing implementations of functions, assertions that cannot be decided during RAC, or by reaching a step limit or time limit (see Section 4.4 below). Finally, the proof failure is categorised according to Figure 11 from the results of the two RAC executions.

5 Experimental Evaluation

We demonstrate our approach by reproducing experiments from prior literature about the categorisation of proof failures [18]. These original experiments were carried out in the C language with ACLS program annotations, and the deductive verification was done using the Frama-C framework. The experiments comprised of a total of 20 modifications to three programs to introduce proof failures, which were then categorised using their approach. The categorisations of the proof failures where obtained by generating two programs that implement RAC in the original program. One generated program implemented the concrete RAC semantics, the other implemented the big-step RAC semantics. Counterexamples on the generated programs where obtained by dynamic symbolic execution. To reproduce the experiments, we translated the programs from C to WhyML, applied the modification (when applicable), and report the results from our RAC approach here. We used the Z3 prover to generate candidate counterexamples, and to prove assertions during RAC.

5.1 Experiment: Integer square root

Figure 21 shows a WhyML function that computes the integer square root. The result for parameter n is an integer r such that $r^2 \leq n < (r+1)^2$. Initially, the variable r contains n as an over-approximation of the result, y contains n^2, and z contains $-2 * n + 1$. 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 is unsatisfied, r contains the largest integer such that $r^2 \leq n$.

Contrary to the original C program, variables written by a loop are inferred by Why3 and do not require explicit annotations in the WhyML program. We further use monomorphic integer reference instead of Why3's standard polymorphic reference type for mutable variables, because counterexample generation in Why3 is not well supported in the presence of polymorphic data types. The original implementation also contained an upper bound of 10,000 for the parameter `n`, which is not required to generate valid counterexamples in Why3.

Splitting the generated verification conditions in Why3 generates ten verification goals for the program: one for the initialisation and preservation of the four loop invariants, one for the variant decrease, and one for the postcondition. The validity of the program is proven in Why3 without further interaction.

Figure 22 shows the ten modifications S1-S10, which were applied to program `isqrt` in the original experiment. Each modification induces a proof failure, for which a counterexample is generated. Using small-step RAC and giant-step RAC, the proof failure is then categorised as a non-conformity of the program (NC) or a subcontract weakness (SW).

S1 Removing the precondition leads to a proof failure for the initialisation of loop invariant I_1. The counterexample provides -1 as value for the function parameter `n`. The concrete RAC proceeds by initialising the variables `r` to -1, `y` to 1, and `z` to 3, but terminates when entering the loop with an assertion failure, because invariant I_1 is invalid for the current variable bindings. The proof failure is categorised as a non-conformity.

S2 Using `2 * n + 1` as the initial value of `z` leads to a proof failure for the initialisation of the loop invariant I_4. The counterexample provides 1 for parameter `n`. Concrete RAC proceeds by initialising the variables `r` and `y` to 1 and `z` to -1, and terminates again with an assertion failure for invariant I_4 when entering the loop, categorising the proof failure as a non-conformity.

S3 Replacing invariant I_4 by `z = 2 * r + 1` leads to a proof failure in the initialisation of invariant I_4. The counterexample provides 1 for parameter `n`, and the concrete RAC fails with an assertion failure on I_4 when entering the loop, categorising the proof failure as non-conformity. (This modification also results in a proof failure for the preservation of invariant I_2. However, the SMT solver fails to

```
1  use int.Int, lib.IntRef
2
3  let isqrt (n: int)
4    requires { 0 <= n }
5    ensures { result * result <= n < (result + 1) * (result + 1) }
6  = let r = int_ref n in
7    let y = int_ref (n * n) in
8    let z = int_ref (-2 * n + 1) in
9    while !y > n do
10     invariant I1 { 0 <= !r <= n }
11     invariant I2 { !y = !r * !r }
12     invariant I3 { n < (!r+1) * (!r+1) }
13     invariant I4 { !z = -2 * !r + 1 }
14     variant { !r }
15     y := !y + !z;
16     z := !z + 2;
17     r := !r - 1
18   done;
19   !r
```

Figure 21: Computation of the integer square root in WhyML

	Line	Substitution	Proof failure	n	r'	y'	z'	Cat.
S1	4	`(* empty *)`	Inv. init. I_1	-1	–	–	–	NC
S2	8	`let z = int_ref (2 * n + 1) in`	Inv. init. I_4	1	–	–	–	NC
S3	13	`invariant { !z = 2 * !r + 1 }`	Inv. init. I_4	1	–	–	–	NC
S4	15	`y := !y - !z;`	Inv. pres. I_2	4	–	–	–	NC
S5	13	`(* empty *)`	Inv. pres. I_2	2	2	4	-4	SW
S6	9	`while !y > n + 1 do`	Postcond.	3	–	–	–	NC
S7	12	`(* empty *)`	Postcond.	1	0	0	1	SW
S8	19	`!r - 1`	Postcond.	0	–	–	–	NC
S9	14	`variant { !r - n }`	Var. decr.	3	–	–	–	NC
S10	10	`invariant { !r <= n }`	Var. decr.	0	-2	4	5	SW

Figure 22: Modifications to the program `isqrt` in Figure 21, with the induced proof failures, relevant values from the counterexample (values for function parameter `n` and for the variables `r`, `y`, and `z` when entering the loop during giant-step RAC execution), and assigned error category for the proof failure.

generate a valid counterexample that is not "shadowed" by a failure in the initialisation of invariant I_4.)

S4 Changing the assignment in line 13 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 small-step 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.

S5 Removing invariant I_4 leads to a proof failure in the preservation of invariant I_2. The counterexample provides the value 2 for the parameter, and the small-step RAC terminates normally with a value of 1. The giant-step RAC queries the counterexample for the initial values for `r`, `y`, and `z` (instead of executing the calls to function `iref`). When performing a big step over the loop, it sets the variables `r`, `y`, and `z` to values 2, 4, and -4 from the counterexample, conforming to the invariants. At the end of the loop body, the variables have been updated to 1, 0, and -2, and the preservation of invariant I_4 fails. Since concrete RAC succeeded and only big-step RAC failed, the proof failure is categorised as a subcontract-weakness.

S6 Changing the loop condition to `!y > n + 1` leads to a proof failure in the postcondition of the function. Executing the function using concrete RAC with an argument of 3 (from the counterexample) returns 2 instead of 1, due to the additional iteration of the loop, and the proof failure is categorised is non-conformity.

S7 Removing the loop invariant I_3 leads to a proof failure in the postcondition. The concrete RAC of the function with argument 1 terminates normally with value 1. After initialisation of the variables, the big-step RAC sets the values of `r`, `y`, and `z` to 0, 0, and 1 respectively according to the counterexample, conforming to the loop invariants. The execution of the loop then terminates, and the function returns 0, which triggers a failure for the postcondition. The proof failure is categorised as a subcontract weakness.

S8 Returning `!r - 1` leads to a proof failure in the postcondition. The function returns in the concrete RAC -1 for argument 0 from the counterexample, which contradicts the postcondition. The proof failure is categorised is non-conformity.

S9 Using `!r - n` as loop variant leads to a proof failure for the loop variant decrease. The counterexample provides the value of 3 for the function parameter `n`. The concrete RAC proceeds the execution until the end of the first loop iteration, where an assertion failure is produced for the decrease of the loop variant, categorising the proof failure as non-conformity.

S10 Changing loop invariant I_1 to `!r <= n` leads to a proof failure for the loop variant decrease. The counterexample provides 0 as parameter `n`, for which concrete RAC terminates normally with 0. When entering the loop, the big-step RAC sets the values of `r`, `y`, and `z` to -2, 4, 5 according to the counterexample. After the execution of the loop body, the values are updated to -3, 9, and 7. The decrease of the loop variant is violated by the negativity of the value `r`. The proof failure is categorised as a subcontract-weakness.

5.2 Experiment: Binary search

The WhyML version the program `binary_search` is shown in Figure 23. It implements a binary search on a sorted array. Again, monomorphic references and arrays are used to obtain counterexamples and, contrary to the original C/ACLS program, written variables are not annotated but inferred by Why3.

In the original experiments there were six modifications B1-B6 to the program `binary_search`, which are shown in Figure 24 together with the results from our analysis where applicable:

```
1  use int.Int, int.ComputerDivision, lib.IntArray, lib.IntRef
2
3  let binary_search (t: int_array) (x: int) : int
4    requires P1 { 1 <= t.length <= 10000 }
5    requires P2 { forall i j. 0 <= i < j < t.length → t[i] <= t[j] }
6    ensures Q1 { -1 <= result <= t.length - 1 }
7    ensures Q2 { forall i. 0 <= i <= result → t[i] <= x }
8    ensures Q3 { forall i. result < i < t.length → t[i] > x }
9  = let l = int_ref (-1) in
10   let r = int_ref (t.length-1) in
11   while !l < !r do
12     invariant I1 { -1 <= !l <= !r < t.length }
13     invariant I2 { forall i. 0 <= i <= !l → t[i] <= x }
14     invariant I3 { forall i. !r < i < t.length → t[i] > x }
15     variant { !r - !l }
16     let m = div (!l + !r + 1) 2 in
17     if t[m] > x
18     then r := m-1
19     else l := m
20   done;
21   !l
```

Figure 23: Binary search in WhyML

	Line	Substitution	Proof failure	t	x	l	r	Cat.
B1	15	`variant { t.length - !r }`	Var. decr.	[17870]	17869	–	–	NC
B2	16	`let m = div (!l + !r) 2 in`	Var. decr.	$[-17869; \ldots]_{57}$	0	–	–	NC
B3	5	`(* empty *)`	Inv. pres. I_3	[3;1;3;3;3]	1	–	–	NC
B4	13, 14	`(* empty *)`	Postcond. Q_3	$[11;\ldots]_5$	11	-1	-1	SW
B5		*not applicable (changes to `written` variables)*						
B6		*not applicable (changes to `written` variables)*						

Figure 24: Modifications to the program `binary_search`, with induced proof failures, relevant values from the counterexample (for function parameters `t` and `x` and variables `l` and `r` in the giant-step execution of the while loop), and the assigned error category. Constant arrays of length n are shown as $[v; \ldots]_n$.

B1 Changing the loop variant to `variant { t.length - !r }` is invalid. The counterexample holds the singleton array [17870] for parameter `t` and 17869 for parameter `x`. The concrete execution fails after the first loop iteration when checking the variant, and the proof failure is categorised as non-conformity.

B2 Removing addend `1` in the computation of `m` results in a proof failure for the variant decrease. The counterexample holds the values `x=0` and `t`=[-17869,...]$_{57}$, a constant array of length 57. The concrete execution fails for the variant decrease, and the proof failure is categorised as non-conformity.

B3 Removing precondition P_2 results in a proof failure for the invariant preservation for I_3. The counterexample value for parameter `t` is an unsorted array. During the small-step RAC execution, loop invariant I_2 holds initially due to the empty quantification, but the preservation fails at the end of the first iteration with `!l = 2`. The proof failure is categorised as non-conformity.

B4 Removing loop invariants I_2 and I_3 results in a proof failure for postcondition Q_3. The counterexample value for the parameter `t` is a constant array of 11s with length 5, and the counterexample value for parameter `x` is 11. The small-step RAC execution terminates normally with value 8. The giant-step execution runs the while loop with counterexample values `l=-1` and `r=-1`, and leaves the loop at the end of the iteration with `l=-1`, which is returned, but contradicts postcondition Q_3. The proof failure is categorised as a subcontract-weakness.

B5 The modification in the original program consisted in changing the annotation for variables written by the while loop. The modification is not applicable, because written variables are inferred and checked during type checking in Why3.

B6 Same as B5.

5.3 Experiment: Restricted growth

The WhyML version of the restricted-growth function is shown in Figure 25. Restricted growth is a property of non-empty arrays, where the value at index 0 is zero, and the value at each positive index increases at most by one over the value the precedent index. The lemma `max_rgf` states that the value at any index is limited by the index. The argument to function `g` is an array with restricted growth and function `g` sets the values at all indices starting at a given index to zero, retaining the restricted-growth property of the array. Function `f` also takes an array with restricted growth as argument and first searches for the last index in an restricted-growth array whose value does not increase over the value of the precedent index. It then uses function `g` to set the values larger than that index to zero. Function `g` retains the restricted-growth property.

In difference to the original C program, the array size is part of the array type and does not have to be passed as argument, and the variables written by function `g` and the loop in `f` are inferred by Why3. The WhyML version again uses monomorphic implementation of arrays and references to facilitate the generation of counterexamples. The lemma `max_rgf` is implemented as a recursive lemma function in WhyML, which is automatically proven in Why3. To facilitate the generation of valid counterexamples, an upper bound of 10,000 had to be introduced for the length of array argument of function `f` (as it was also done in the original experiment for programs `isqrt` and `binary_search`).

The original experiment applied four modifications R1-R4, which are shown in Figure 26:

R1 Removing precondition P_1' introduces a proof failure for precondition P_2 of the call to `g` in line 41. The counterexample provides an array of length 9824 that doesn't conform to the restricted growth property, and the concrete execution fails for the precondition. The proof failure is categorised as non-conformity.

```
1  use int.Int, lib.IntArray, lib.IntRef
2
3  predicate is_rgf (a: int_array) (n: int) =
4    a.length > 0 ∧ a[0] = 0 ∧ forall i. 1 <= i < n → 0 <= a[i] <= a[i-1]+1
5
6  let rec lemma max_rgf (a: int_array) (n: int)
7    requires { is_rgf a n }
8    requires { 0 <= n < a.length }
9    ensures { forall i. 0 <= i < n → a[i] <= i }
10   variant { n }
11 = if n > 0 then max_rgf a (n - 1)
12
13 let g (a: int_array) (i: int)
14   requires P1 { 1 <= i <= a.length }
15   requires P2 { is_rgf a (i+1) }
16   ensures { forall j. 0 <= j <= i → (old a)[j] = a[j] }
17   ensures { is_rgf a a.length }
18 = for k = i + 1 to a.length - 1 do
19     invariant { is_rgf a k }
20     invariant { forall j. 0 <= j <= i → (old a)[j] = a[j] }
21     a[k] ← 0
22   done
23
24 let f (a: int_array) : bool
25   requires { 0 < a.length < 10000 }
26   requires { is_rgf a a.length }
27   ensures { is_rgf a a.length }
28   ensures { result = True →
29     exists j. 0 <= j < a.length ∧
30     (old a)[j] < a[j] ∧
31     forall k. 0 <= k < j → (old a)[k] = a[k] }
32 = let i = int_ref (a.length - 1) in
33   while !i >= 1 do
34     invariant { 0 <= !i <= a.length - 1 }
35     variant { !i }
36     if a[!i] <= a[!i-1] then break;
37     i := !i - 1
38   done;
39   if !i = 0 then return False;
40   a[!i] ← a[!i] + 1;
41   g a !i;
42   assert { forall j. 0 <= j < !i → (old a)[j] = a[j] };
43   return True
```

Figure 25: Restricted growth for arrays in WhyML

	Line	Substitution	Proof failure	a	Cat.
R1	26	`(* empty *)`	Precond. P_2 in line 41	[17869; -400; 8; . . .]$_{9824}$	NC
R2		*not applicable (changes to* `written` *variables)*			
R3		*not applicable (removed lemma* `max_rgf`*)*			
R4	40	`a[!i] ← a[!i] + 2;`	Precond. P_2 in line 41	[0; 0]	NC

Figure 26: Modifications to the program implementing restricted growth, with the induced proof failures, relevant counterexample values (for the function parameter `a`) and the assigned error category.

R2 The modification in the original program consisted of changing the list of variables written by the while loop. The modification is not applicable because written variables are inferred by Why3.

R3 The modification in the original program consisted of removing lemma `max_rgf`, which introduced a prover incapacity for an assertion that states the absence of an integer overflow. In the WhyML version of program can be proven without the lemma. (The lemma, however, is required to generate a valid counterexample for R4.)

R4 Increasing the selected array values by 2 instead of 1 in line 40 breaks the restricted growth property of the array. The counterexample value for parameter `a` is the array [0; 0]. The array is modified in line 40 to [0; 2], and precondition P_2 fails for the call in line 41. The counterexample is categorised as non-conformity.

6 Conclusions, Discussions, Related Work and Future Work

We proposed an approach for validation and categorisation *a posteriori* of a candidate counterexample that was itself derived [8] from a model proposed by an SMT solver. First, the analysis starts by using the candidate counterexample to extract values for the global variables and also for the parameters of the target function. These values are then used to run a standard small-step RAC. Second, again from candidate counterexample, we extract a set of values for variables that are modified by function calls and loops and run a novel *giant-step* variant of RAC. This variant mimics the form of the verification condition that is generated for function calls and loops in the weakest precondition calculus. The different outcomes of standard RAC and giant-step RAC allow us to categorise the candidate counterexample.

A natural question would be how this approach would be affected if the generation of the VC was based on some variants of the WP calculus, such as the ones that are considered more efficient in the sense that they generate formulas which are not exponential in the size of the code [11, 13, 15, 2]. In fact we believe our approach is not affected, because all those calculi treat function calls and loops in a similar manner, producing a formula quantified over the variables modified. In fact a common approach is to get rid of iteration and function calls before VC generation [11, 2].

6.1 Related Work

The closest related work are those of Christakis et al.[5] and of Petiot et al. [17, 18]. They have the same goal as us, that is to provide useful feedback to the user when a proof fails. Generally speaking, they differ from our approach in the sense that instead of trying to validate a model returned by an SMT solver, they use some other method for providing a test case that leads to a the considered proof failure.

In [5], the authors use Dynamic Symbolic Execution (DSE) (also known as concolic testing) to generate test cases for the part of the code that originated the VC. The code is instrumented with run-time checks and then fed to Delfy, a dynamic symbolic executor, that will explore all possible paths up to a given bound. The output can be one of the following: the engine is able to verify the method, indicating that the VC is valid and thus no further action is required; a test case that leads to an assertion violation, indicating that the VC is invalid; cannot generate a test case for the given bound and thus nothing can be concluded.

Our classification of counterexample is based on those of Petiot et al. [17, 18]. Also here, the authors rely on DSE, more precisely on PathCrawler, to classify proof failures and to generate counterexamples. For each proof failure they classify it as *non-compliance*, *sub-contract weakness*, or *prover incapacity*. Note that since they rely on DSE, any counterexample will lead to a violation when executing the program concretely (they have no "bad counterexample").

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 a DSE engine. Instead of applying different program transformations, we compare the results of two different kinds of executions (standard small-step RAC and giant-step RAC) of the original program.

6.2 Future Work

Technical issues. In the future, our approach should be extended to support more features. Verification tools such as Why3 support *type invariants* and VCs are normally generated to prove their preservation. For the time being our implementation does not support them. Another technical limitation is that when the original code makes use of polymorphic data types, a more complex encoding needs to be applied to the VCs before sending them to the provers. At the time of this publication our implementation is not able to unwind this encoding to a obtain an oracle for running the RAC. Also, as seen in the experiments, our

method is often limited in the RAC due to the fact that the annotation language is not always executable. This is also a known issue for RAC [12].

Identifying the single subcontract weaknesses. Our approach can classify a CE as a subcontract weakness, but it does not point out where is the weakness. It is of course desirable to improve the localisation of the weakness. An idea could be as follows. Consider a program where a subcontract weakness has been detected, *i.e.* the small-step execution terminated normally and the giant-step execution failed. Let ϕ be the post-condition of a function f or the invariant of a loop l. The program contains a *single sub-contract weakness* in ϕ, if a mixed giant-step execution, where only function f or loop l are executed with small-step semantics, terminates normally. Detecting a single sub-contract weakness allows for pointing to a specific function or loop whose contracts should be strengthened to resolve the proof failure. Note that the absence of single subcontract weakness does not imply the absence of global subcontract weakness [18].

Dealing with incomplete information. In practice we sometimes have to cope with some lack of data. An example is when the code contains functions without implementation, or it uses the `any` construct which is a kind of non-deterministic execution. An idea that we implement is to use the giant-step semantics even in the standard RAC for those. Yet, this does not provide satisfactory results, in particular it does not work well within a loop body: the oracle is certainly not able to provide values for written variables if there is a function call that takes place inside the loop.

Another source of lack of information is the possibility of incomplete oracles. Indeed, values that are required for the small-step or giant-step RAC may be missing from the SMT solver model. In this case, the model could be considered incomplete, the RAC execution terminates as incomplete, and the counterexample cannot be categorised. However, in many cases the RAC engine can sidesteps to other strategies to generate the values. We have implemented such strategies, but these can certainly be improved, for example using randomised techniques discussed below. In any strategy, care has to be taken to generate only values that don't cause failures. Otherwise the counterexample is blamed to contain invalid values and considered bad instead of just being incomplete.

Support by front-ends. As mentioned in the section about experiments, our implementation aims to support various Why3 front-ends, such the one for Ada/SPARK [16]. 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 it might be necessary to perform the small-step assertion checking in the front-end language to obtain more accurate diagnoses. Yet, the giant-step execution should still be done at the level of Why3 since it is related to how the VC are generated.

Combination with other techniques. In a longer term, it seems worth investigating the combination of our approach with other techniques, such as the DSE mentioned above, and randomised approaches such as fuzzy testing or quick check.

Acknowledgements. We thank our partners from AdaCore (Yannick Moy) and MERCE (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, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2032305.2032319.

[2] Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. A Generalized Program Verification Workflow Based on Loop Elimination and SA Form. In *FormaliSE 2019 - 7th International Conference on Formal Methods in Software Engineering*, Montreal, Canada, May 2019. URL: https://hal.inria.fr/hal-02431769.

[3] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/.

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

[5] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. Integrated environment for diagnosing verification errors. In *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*, pages 424–441. Springer, 2016.

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

[7] John Coltrane. Giant steps. https://en.wikipedia.org/wiki/Giant_Steps_(composition), 1959. See also [19].

[8] Sylvain Dailler, David Hauzar, Claude Marché, and Yannick Moy. Instrumenting a weakest precondition calculus for counterexample generation. *Journal of Logical and Algebraic Methods in Programming*, 99:97–113, 2018. URL: https://hal.inria.fr/hal-01802488.

[9] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

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

[11] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Principles Of Programming Languages*, pages 193–205. ACM, 2001.

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

[13] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

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

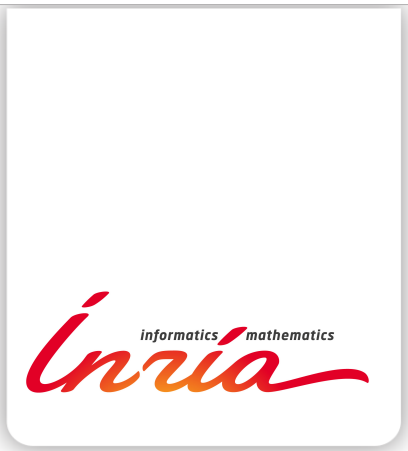
[15] Cláudio Belo Lourenço, Si-Mohamed Lamraoui, Shin Nakajima, and Jorge Sousa Pinto. Studying verification conditions for imperative programs. *Electronic Communication of the European Association of Software Science and Technology*, 72, 2015. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/1011.

[16] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[17] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. Your proof fails? testing helps to find the reason. In *Tests and Proofs - 10th International Conference*, volume 9762 of *Lecture Notes in Computer Science*, pages 130–150. Springer, 2016.

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

[19] Sting. Walking on the moon. https://en.wikipedia.org/wiki/Walking_on_the_Moon, 1979. See also [7].



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399