Counterexamples from proof failures in the SPARK program verifier

David Hauzar, Claude Marché, Yannick Moy

Counterexamples from proof failures in the SPARK program verifier

David Hauzar, Claude Marché, Yannick Moy

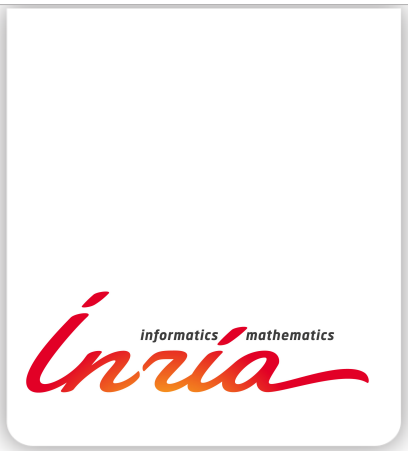
RESEARCH REPORT

N° 8854

February 2016

Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--8854--FR+ENG



Counterexamples from proof failures in the SPARK program verifier

David Hauzar*, Claude Marché*†, Yannick Moy‡

Project-Team Toccata

Research Report n° 8854 — February 2016 — 22 pages

Abstract: A major issue in the activity of deductive program verification is the understanding of the reason for why some proof fails. To help the user understand the problem and decide what needs to be fixed in the code or the specification of her program, it is essential to provide means to investigate such a failure. To that mean, we propose a technique for generating *counterexamples*, exhibiting some values for the variables of the program where a given part of the specification fails to be validated.
To produce such a counterexample, we exploit the ability of SMT (Satisfiability Modulo Theories) solvers to propose, when a proof of a formula is not found, a *counter-model*. Turning such a counter-model into a counterexample for the initial program is not a trivial task because of the many transformations that lead from a given code and specification to a verification condition. We report on our approach for the design and the implementation of counterexample generation within the SPARK 2014 environment for the development of safety-critical Ada programs.

Key-words: Formal Specification, Deductive Verification, Formal Proof, Counterexamples, Program Verifiers Why3 and SPARK

* Inria, Université Paris-Saclay, F-91893 Palaiseau
† LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay
‡ AdaCore, F-75009 Paris

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

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

Contre-exemples issus d'échecs de preuve dans l'environnement SPARK

Résumé : Un défi important de l'activité de vérification déductive de programmes est la compréhension de la raison pour laquelle une certaine preuve échoue. Pour aider l'utilisateur à comprendre le problème et à décider quoi modifier dans son code ou bien sa spécification, il est essentiel de fournir des moyens d'analyser l'échec de preuve. À cette fin, nous proposons une technique de génération de *contre-exemples*, qui exhibent des valeurs pour les variables du programme pour lesquelles une certaine partie de la spécification ne peut pas être prouvée.

Pour produire un tel contre-exemple, nous exploitons la capacité des solveurs SMT (Satisfiability Modulo Theories) de proposer, quand une formule n'est pas prouvée, un *contre-modèle*. Transformer un tel contre-modèle en un contre-exemple pour le programme de départ n'est pas une tâche facile à cause des multiples transformations qui conduisent du code et de la spécification de départ vers une obligation de preuve. Nous présentons l'approche que nous avons suivie pour la conception et l'implémentation de la génération de contre-exemples dans le cadre de l'environnement SPARK 2014 pour le développement de programmes Ada critiques.

Mots-clés : Spécification formelle, preuve de programmes, contre-exemples, environnements de preuves Why3 et SPARK

Contents

1 Introduction

Deductive program verification is an activity that aims at checking that a given program respects a given functional behavior. In this context, the expected behavior must be expressed formally by logical assertions, e.g. preconditions and postconditions, forming a *contract*. Deductive program verification typically proceeds by generating, from both the code and the formal specification, a set of logic formulas called *verification conditions* (VCs). If one proves that all generated VCs are tautologies, then the program is guaranteed to satisfy its specification.

In recent program verification environments like Dafny [18] and Why3 [7], verification conditions are discharged using automated theorem provers, in particular those of the *Satisfiability Modulo Theories* (SMT) family such as Alt-Ergo [5], CVC4 [2] and Z3 [12]. These theorem provers are used as black-boxes that, given a VC, may produce 3 kinds of results:

1. The prover answers something meaning "yes, the VC is a tautology"
2. The prover answers anything else, meaning "I don't know", in order words the prover is not able to prove the VC for any reason
3. The prover runs for a too long time (seemingly infinitely) or runs out of memory

The first case is naturally the most favorable one, and when all VCs are proved valid then the program can be considered as proved sound with a very high level of confidence. The case where the prover runs for too long time is handled in practice by setting a given time limit, so that the prover process is killed when exceeding this limit. The cases 2 and 3 are then somehow the same: the VC is not proved. Note that we do not distinguish a case where the prover would answer "no it is not a tautology", because the VCs typically involve undecidable logic features (e.g. non-linear arithmetic, first-order quantification) so provers are by essence incomplete: there is no way for them to be sure that a given VC is not provable.

A major issue in the activity of deductive verification is thus understanding the reasons for a proof failure. There are various reasons why it may fail:

- The property to prove is indeed invalid: the code is not correct in the sense that it really does not respect the given specification.
- The property is in fact valid, but is not proved, again for 2 possible reasons:
  - The prover is not able to obtain a proof (in the given time and memory limits): this is the incompleteness of the proof search;
  - The proof may need extra intermediate annotations, such as loop invariants, or more complete contracts of the subprograms called: this is the incompleteness of the VC generation process.

For the user to be able to fix the code or the specification of its program, it is essential to provide means to investigate a failure, to understand in which case the reason lie, and to help fixing the problem. This is the main goal of this paper. The solution we propose to this problem is to generate *counterexamples*, or more precisely *potential* counterexamples. Such a counterexample should propose values for the variables of the program, exhibiting a particular execution where a given annotation may fail.

To produce such a counterexample, we want to exploit an additional feature of SMT solvers: the ability to propose, when a proof of a formula is not found, a *counter-model*, exhibiting an interpretation of the free variables where the formula does not seem to hold. Turning such a counter-model into a counterexample for the initial program is not a trivial task because of the many transformations that lead to a VC from a given code and specification. For this work, our goal was to design and implement counterexample generation within the SPARK 2014 [19] environment for the development of safety-critical Ada programs. In this context, the initial program with annotations is first translated into the intermediate

programming language WhyML. The Why3 tool [7] processes WhyML code to generate VCs using a weakest precondition calculus. These VCs are then passed to SMT solvers after several possible transformations: simplifications and encodings of features not natively supported by SMT-LIB. Then, to turn the counter-model into a counterexample, one has to somehow revert back all the transformation chain.

We start in Section 2 by giving a quick overview of SPARK 2014. In Section 3 we present the support we introduced in SPARK 2014 for counterexamples, from a user's point of view, illustrated by simple examples. In Section 4 we go into the internals of the tools, and explain how we designed our approach to generate counterexamples. We discuss related work and future work in Section 5.

2 SPARK 2014 in Brief

Ada is a programming language targeted at real-time embedded software which requires a high level of safety, security, and reliability. In particular, it provides a wide range of checks for run-time errors, for example for buffer overflows, and has a verbose syntax that makes it easy to read and debug. For these reasons, Ada is nowadays used in domains where software cannot be allowed to fail .

Ada 2012 is the latest version of the Ada language [1]. With respect to the former version Ada 2007, it contains new features for specifying the expected behavior of programs, such as subprogram contracts and type invariants. When given a specific compilation switch, the Ada compiler can turn these constructs into assertions to check at run time. Thanks to this switch, the conformance of the implementation of a program to its specification can be checked dynamically during the process of unit testing.

SPARK, co-developed by Altran and AdaCore, is a subset of Ada targeted at formal verification. Its restrictions ensure that the behavior of a SPARK program is unambiguously defined (unlike Ada). It excludes constructions that cannot easily be verified by automatic tools. The SPARK language and toolset for static verification has been applied for many years in on-board aircraft systems, control systems, cryptographic systems, and rail systems [8].

SPARK 2014, the latest version of SPARK, is a subset of Ada 2012 [19]. It comprises most of the Ada 2012 language excluding constructs which are not easily amenable to sound static verification: features such as pointers, side effects in expressions, aliasing, goto statements, controlled types (e.g. types with finalization) and exception handling are excluded.

SPARK 2014 is designed so that both the flow analysis (checking that there is no access to uninitialized variables and that global variables and subprogram parameters are accessed appropriately) and the program's proof (checking the absence of run-time errors and the conformance to the contract) can be performed. It provides dedicated features that are not part of Ada 2012. In particular, contracts can also contain information about data dependencies, information flows, state abstraction, and data and behavior refinement that can be checked by the GNATprove tool. Essential constructs for formal verification such as loop variants and invariants have also been introduced.

As described in Figure 1, to formally prove a SPARK 2014 program, GNATprove uses WhyML as an intermediate language. The SPARK program is translated into an equivalent WhyML program which can then be verified using the Why3 tool.

Figure 2 shows an example of a saturation procedure. Saturation procedure is used to ensure that values are in a given range. In this case, the saturation procedure should ensure that the output value of its non-negative argument is less or equal to 255. More precisely, the postcondition of the procedure requires that if the input value of the argument is in the range, the value of the argument is not modified and if the input value of the argument is out of the range, the output value is 255—the closest value to the input value that is in the range. Note that an attribute `Old` in the postcondition of a procedure refers to values that expressions had at entry. The procedure is implemented using bitwise and of the input value with a mask `16#FF#` (`0000000011111111`). However, as the message at the bottom of Figure 2 shows, GNATprove does not succeed in proving the postcondition.

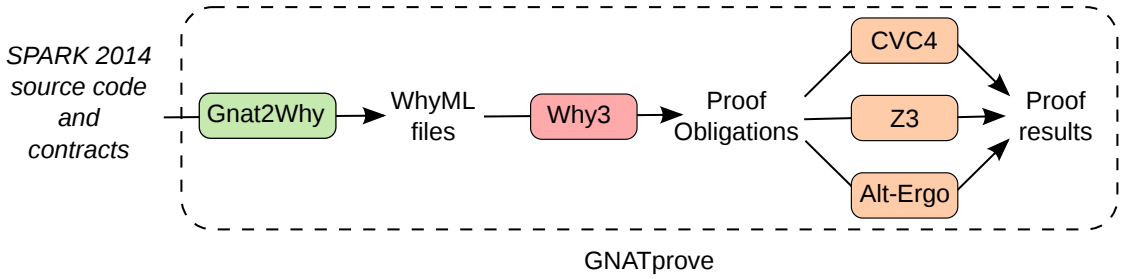


Figure 1: Deductive verification in SPARK 2014

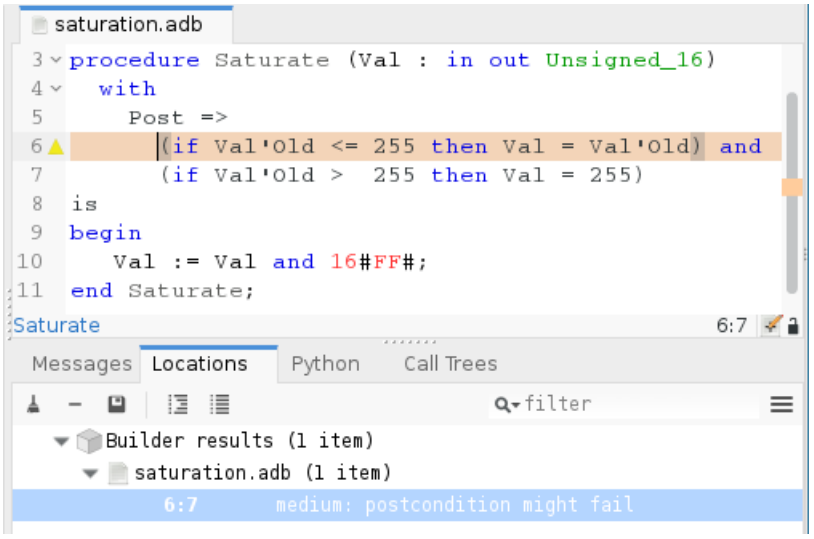


Figure 2: Saturation procedure with failed postcondition.

The verification process in SPARK is based on interactions between the user and the verification environment. Whenever GNATprove does not succeed in proving a property, it indicates the line of the property that is not proved, with a simple message like the one of Figure 2: "postcondition might fail".

The means for the user to investigate the possible reason of the failure are:

- Execute code and properties during tests, in a way that violations of the property will stop execution with an exception. This depends of course on the availability of tests that exercise the violation, but testing is a well-known software engineering discipline that engineers usually master, hence uncovering incorrect code and properties is comparatively easier than investigating other reasons for proof failure.
- A focused manual review of the code and assertions can efficiently diagnose many cases of missing annotations.
- The user can try to increase the proof power along different axes, in order to combine the results of different provers and allocate more resources (in particular time) for each proof attempt. In GNATprove, in additions to the lower level switches, we have predefined *proof levels* between 0 and 4 that the user can increase to augment the proof power: more time allocated, use more provers.

Also, GNATprove helps users by pinpointing the part of a larger assertion which is not proved, and the execution path along which the proof fails. During interactions, the IDE integration is of utmost importance to allow focusing the proof on a single subprogram or even a single line of code. Yet, manual reviews may not allow to identify missing annotations, and increasing the proof power may not allow to prove the property. In such a case, the burden is on the user to verify the unproved property by other means, either using tests, or manual reviews, or a manual prover whose proof script is checked by GNATprove.

3 Adding Counterexamples in SPARK

There are multiple ways to integrate counterexamples in a development environment, depending on the expected degree of interactions with users. In SPARK, we have chosen to simplify the interactions to a minimum, so that users are directly presented with the most relevant information. On the one hand, other choices would have allowed more flexible uses of counterexamples. On the other hand, we believe this choice of simplicity facilitates the life of our users.

By default, GNATprove displays the values of relevant variables in the message displayed to the user for an unproved check. For example, here is the message displayed by GNATprove on the example from Figure 2 seen before:

```
medium: postcondition might fail (e.g. when Val'Old = 4096 and Val = 0)
```

This information alone might be sufficient to understand the problem. Otherwise, GNATprove has precomputed for every unproved check a counterexample trace that can be displayed in the IDE by simply clicking on the *magnify* icon on the side of the message or on the side of the corresponding line in the editor. This trace consists in a sequence of program lines with values of relevant variables on that line. For example, Figure 3 shows the trace computed by GNATprove and displayed in Gnat Programming Studio on the example seen before.

A variable is selected as *relevant* in the message if it appears in the expression being checked. A variable on a given line is selected as *relevant* in the trace if it is assigned a new value on this line.

As visible from the Figure 3, the counterexample trace is displayed inside special lines in the editor (in gray in the screenshot), that are not part of the code and cannot be edited manually (note the absence of a line number). These lines are prefixed with the token `--` that introduces comments in Ada code to

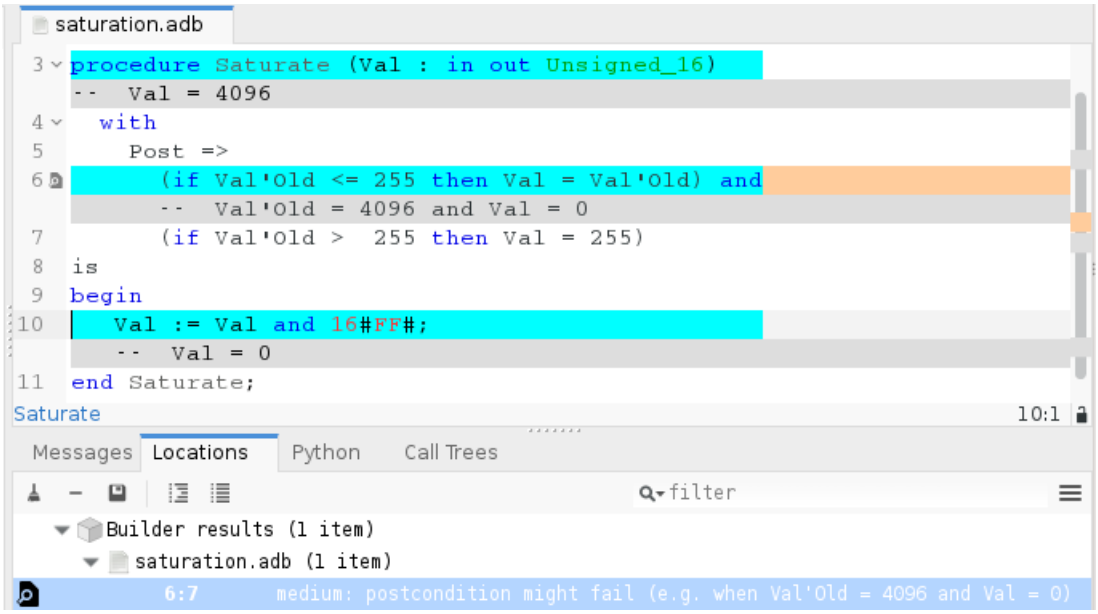


Figure 3: Saturation procedure together with counterexample demonstrating that the implementation does not conform to the postcondition.

make it clear to users that they are not part of the code. The lines in the program to which the trace applies are emphasized in light blue.

The counterexample in Figure 3 shows that the implementation is indeed not correct with respect to the specification. Bitwise and of `4096` (`1000000000000` in binary) and `16#FF#` (`0000011111111` in binary) is `0`, while the specification requires value of `Val` at the end of the procedure be `255`.

3.1 Displaying Counterexamples with Records

Counterexamples can contain values of record types. The values of record types are displayed in the usual Ada syntax as record aggregates. This is illustrated by Figure 4. The values of fields that are not relevant for the counterexample are replaced by question marks. If there are more than one field in the value of record type that are not relevant for the counterexample, they are aggregated under the name `others`.

On Figure 4, type `Saturable_Value` defined at line 5–8 contains a field `Value` representing the actual value and a field `Upper_Bound` being an upper bound of the saturation range. The postcondition of the function `Saturate` is analogous to the postcondition of the procedure `Saturate` from Figure 3 . The field `Value` of the returned record must contain the value of the field `Value` of the input record if it is in the range, otherwise it must contain the upper bound of the range. Instead using bitwise and, the saturation is now implemented using function `Unsigned_16'Max`. The counterexample shows that if `Val.Value` is `16383` and `Val.Upper_Bound` is `49152`, `Saturate'Result.Val` is `49152`. Indeed, instead of the function `Unsigned_16'Max`, the function `Unsigned_16'Min` should be used.

3.2 Displaying Counterexamples with Arrays

Values of array types can also be displayed in counterexamples. Figure 5 shows an example of a function that takes a value and an array of values as arguments, divides the value with each value in the array and

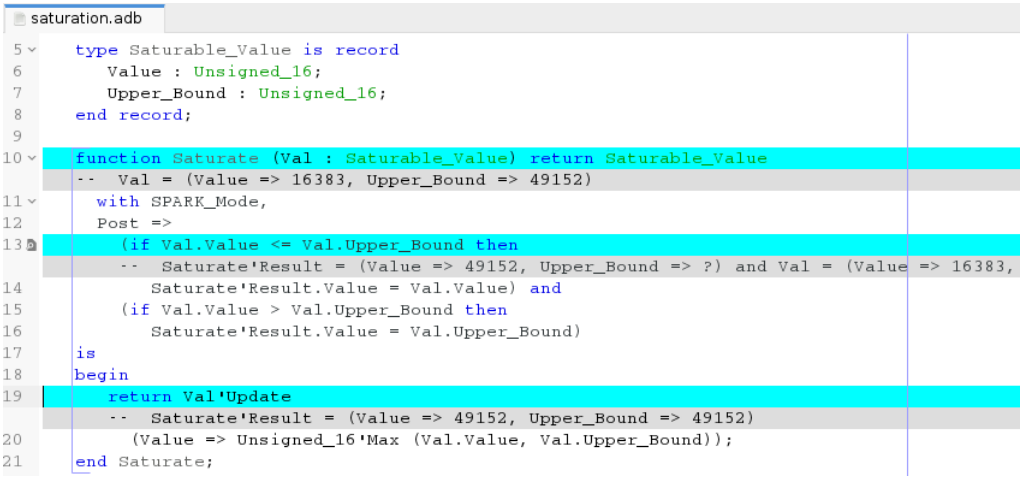


Figure 4: Saturation function with variable saturation range with bug in the implementation. Counterexample shows when the postcondition fails and includes values of a record type.

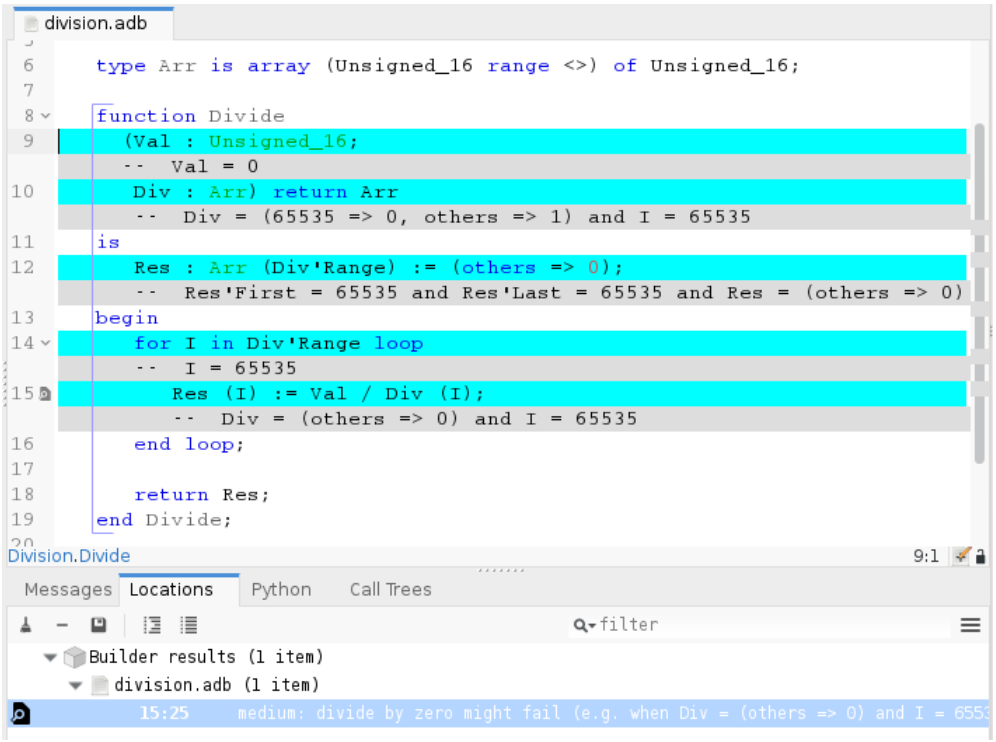


Figure 5: Function that returns the result of division of a number given as an argument `Val` by array of numbers specified as an argument `Div`. The counterexample shows when runtime error can happen.

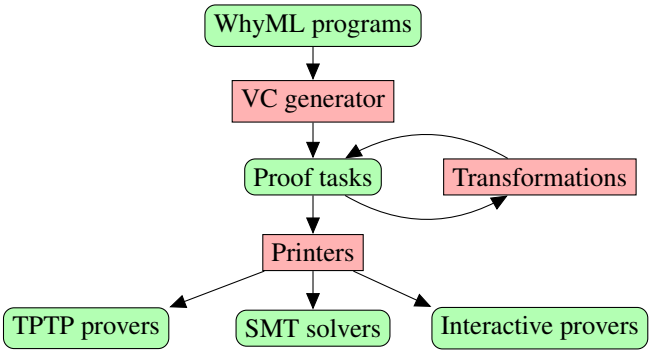


Figure 6: Why3 architecture

returns an array with results of divisions. As shown in the counterexample, division by zero can happen when some value stored in the array is zero.

As for records, array values in counterexamples are displayed in Ada syntax, as array aggregates. The indexes that are not relevant are summarized under the name `others`. In some cases, the array range is also part of the counterexample, shown again in Ada syntax using the attributes `'First` and `'Last`.

4 Implementation of Counterexamples

4.1 Short Introduction to Why3

Why3 is an environment for deductive program verification, providing a rich language for specification and programming, called WhyML. Verification proceeds by generating verification conditions thanks to a weakest precondition calculus. It relies on external provers, both automated and interactive, in order to discharge these verification conditions. WhyML is used as an intermediate language for verification of SPARK programs and also of C and Java programs [13], and is also intended to be comfortable as a primary programming language.

A schematic view a Why3's components is shown on Figure 6. The specification component of WhyML [6], used to write program annotations and background theories, is an extension of first-order logic. It features ML-style polymorphic types (prenex polymorphism) ; algebraic datatypes (in particular records) ; abstract types, functions and predicates specified axiomatically. Why3 comes with a rich standard library providing general-purpose theories useful for specifying programs [7]. This includes integer and real arithmetic, arrays, bitvectors.

The specification part of the language can serve as a common format for theorem proving problems, *proof tasks* in Why3's jargon, suitable for multiple provers. The Why3 tool generates proof tasks from lemmas and goals, then dispatches them to multiple provers, including SMT solvers Alt-Ergo, CVC4, Z3; TPTP first-order provers E, SPASS, Vampire; interactive theorem provers Coq, Isabelle and PVS.

The programming part of WhyML is a dialect of ML with a number of restrictions to make automated proving viable. WhyML function definitions are annotated with pre- and post-conditions both for normal and exceptional termination, and loops are also annotated with invariants. We refer to Filliâtre and Paskevich [14] for more details on the programming part of WhyML. Why3's web site[1] also provides a extensive tutorial and a large collection of examples.

We detail below a few features of Why3 that are of particular interest for the counterexamples feature.

[1] `http://why3.lri.fr`

Transformations A Why3 transformation is any procedure taking a proof task as an argument and producing another proof task, or more generally a set of proof tasks. Transformations must be *sound* is the sense that validity of the resulting tasks must imply the validity of the input task. The converse is generally true but not always. A typical example is the *split* transformation: for a given proof task of the form

$$H_1, \ldots, H_k \vdash \forall \vec{x}.H \rightarrow (G_1 \wedge \cdots \wedge G_n)$$

that is if the goal ends with a conjunction, it produces the set of n tasks

$$H_1, \ldots, H_k \vdash \forall \vec{x}.H \rightarrow G_i$$

for $1 \leq i \leq n$.

As most of the provers do not support some of the language features, (e.g. pattern matching, polymorphic types, recursion), Why3 applies a series of encoding transformations to eliminate unsupported constructions before dispatching a proof task to provers. Other transformations can also be imposed by the user in order to simplify the proof search: inlining of definitions, simplification by computation, case analysis, application of inductive schemes, etc.

Labels Why3 *labels* are arbitrary character strings, written between double quotes. They can be attached to any logic formula or term, and also to any declaration. Their interpretation is not fixed a priori, in some cases they are interpreted by some transformations. For example, the *asymmetric conjunction* of Why3's logic is a connective written as `&&`. Internally, it is in fact the usual conjunction `/\` with the label `"asym_split"` on the first argument. The `split` transformation interprets this label so that a goal of the form `f1 && f2` is split into the goals `f1` and `f1` $\rightarrow$ `f2`. Usually, transformations do not interpret labels and keep them attached to formulas and terms as much as possible. For example, a transformation may rename a variable, in that case it should propagate labels from the original variable to the new one. Analogously, if a transformation rewrites a given sub-term into another, it should also transports labels of the old term to the new one.

Locations To help traceability of errors from its various front-ends, Why3 input language proposes a mechanism of source locations similar to the `#line` directive of C preprocessor. Instead of being line-oriented, it is character-precise: any term or declaration can be given an annotation of the form `#file` l b e`#` meaning that this term or declaration originates from the source file *file*, at line l, from first character b to last character e.

The Weakest Precondition Calculus The VC generator, that implements some variant of the Weakest Precondition calculus (WP for short), takes any WhyML function and creates a proof task. If that proof task is a tautology then the input function satisfies its contract. This formula is typically a quite large formula that collects all the necessary checks that need to hold for the function to be safe: post-condition but also initialization and preservation of loop invariants if any, any kind of runtime checks, etc. To present the resulting formula to the user in a more friendly manner, a default application of the `split` transformation is applied, so as to obtain a set of VCs that corresponds to the various checks to perform on the input code. To make this even more user-friendly, Why3's WP calculus is instrumented so that each of the sub-formulas that corresponds to a program check is annotated with a label of the form `"expl:`*text*`"`. The *text* is an explanation of the VC, and is interpreted by the back-end graphical interface.

Regarding the counterexample feature, an important aspect is that during the computation of the WP, for every program statement that does a side-effect, that is it stores a new value to a program variable, a fresh logical variable holding this new value is created. This is the case for assignment statement, but also occurs in case of function calls, function declaration, and in presence of loops.

Metas Why3's *metas* provide a way to associate metadata to a proof task that, unlike labels, are not attached to any particular sub-term or declaration, but are declared globally to the task. A meta is characterized by a name and a set of parameters that can be nearly of any kind of object: a number, a boolean, a string, but also a reference to another declaration: a type, a function symbol, an hypothesis. As for labels, metas can be interpreted by transformations, but are usually kept unchanged.

Unlike labels, the name of metas, and the type of their arguments, must be declared first, and this can be done only programmatically via Why3 API.

4.2 Model Features of SMT-LIB

An SMT solver takes as input a set of formulas, and checks whether this set is satisfiable or not. To prove that a given proof task $H \vdash G$ is a tautology, we query the solver for the satisfiability of H and the negation of G: if the solver answers that this set is unsatisfiable, it means that proof task is valid. If the solver terminates with any other answer, the SMT solver may propose a potential model of H and $\neg G$ describing why $H \vdash G$ cannot be proved.

To get such a model, we use features of SMT-LIB [3], and the solvers CVC4 and Z3. SMT-LIB defines commands `get-model` and `get-value` for getting models. The command `get-model` returns a set of interpretations for all user-declared function symbols in the input task. The command (`get-value` $t_1 \cdots t_n$) returns for each term t_i a value term that is equivalent to t_i in the potential model. For example, when given the file

```
(set-logic AUFLIRA)
(set-option :produce-models true)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (and (<= 5 (+ x y)) (<= x 3) (<= y 3)))
(check-sat)
(get-value (x y))
```

CVC4 returns

```
sat
((x 3) (y 2))
```

If the `get-value` command is only given `x` as argument, then the solver only provides a value for `x`:

```
sat
((x 3))
```

Notice that we talk about *potential* model here, because we typically query the solvers in logic fragments on which they can cannot be guaranteed to be complete, because of first-order quantifiers, non-linear arithmetic, etc.

4.3 Counterexamples at Why3 Level

Our goal is thus to exploit the generation of models by SMT solvers to construct a potential counterexample to the input Why3 program. This means that we need to add counterexample generation to the Why3 architecture described in Figure 6: some feedback from the bottom—prover results— to the top—input program— must be implemented. Because of the VC generation and the Why3 transformations can rename variables and introduce fresh ones, it is non-trivial process to re-interpret the model returned by the solver into a counterexample of the input source. This is exactly where we need to exploit the labels of Why3.

A first choice we have to make is on whether using the `get-model` or the `get-value` command of SMT-LIB. The command `get-model` might seem easier to use at first because no argument needs to be given. However, from the large set of function symbols and their values returned by `get-model`, it would

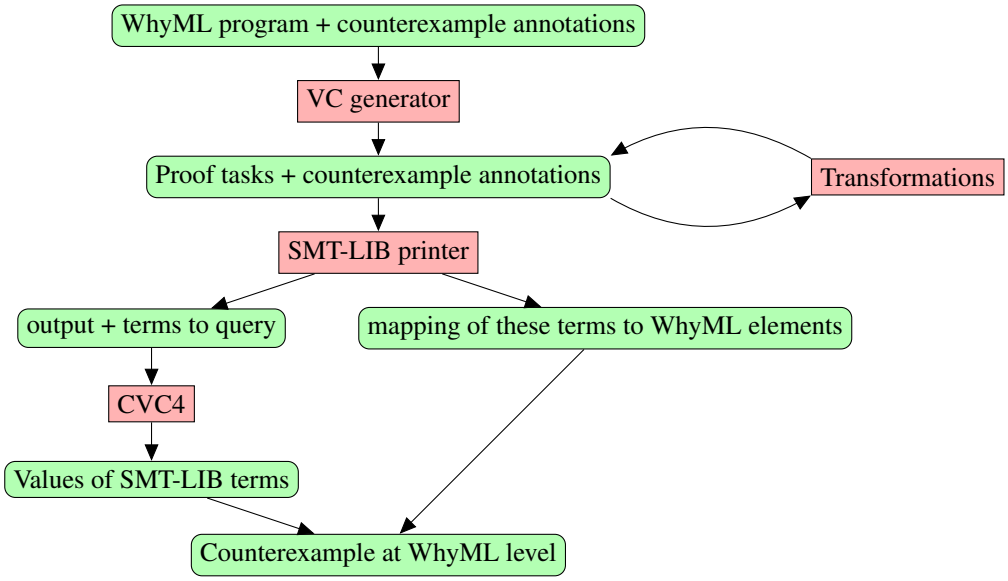


Figure 7: Counterexamples at Why3 level.

be a hard task to extract which part of it corresponds to the initial program, because we have no trace of the extra logical variables and renamings made by WP and transformations. That's why we decided to use the `get-value` command instead. We provide the variables or terms to query as arguments of this command by properly propagating traceability information along the WP and the Why3 transformations. This is done using Why3 labels and metas instrumenting the different processing steps as shown in Figure 7. This has to be performed regarding different aspects that are detailed in the subsections below.

4.3.1 Marking Variables to Show in a Counterexample

In a Why3 task, variables that should be shown in a counterexample are marked with the label `"model"`. Why3 transformations that transform the task must propagate these labels. When the task is transformed into SMT-LIB format, SMT-LIB terms corresponding to these variables are collected and then passed as parameters of the `get_value` command.

As an example, see the following Why3 task:

```
constant x "model" :int
goal G : x+x > 0
```

When printing the task into SMT-LIB formula, the SMT-LIB term corresponding to the constant `x` will be collected and queried for counterexample value `v`.

The counterexample will be displayed to the user in the form `x = v` and this equality will be associated to the location of the goal G. For a Why3 task that is generated from WhyML or SPARK program, we additionally need to annotate each variable with two things. First, with a location in the original source code and second with the name of the variable in the source language. [2]

```
constant x "model" "model_trace:X" #file.adb 42 1 2#:int
goal G : x+x > 0
```

In such a case the counterexample will be displayed in the form `X = v` and associated with location in file `file.adb`, line `42`.

[2] In practise, it is displayed in a comment as shown, e.g., in Figure 3.

```
1  module M
2    use import ref.Ref
3    use import int.Int
4
5    let test_loop ( x "model" "model_trace:X" : ref int ): unit
6    requires { !x > 0 }
7      =
8      while !x > 0 do            (* counterexample: X = 1 *)
9      invariant { !x >= 0 }
10       x := !x - 2              (* counterexample: X = -1 *)
11     done
12 end
```

Figure 8: Example of WhyML program annotated with counterexample labels. . A WhyML counterexample is shown in comments.

```
1  goal WP_parameter_test_loop "expl:VC for test_loop" :
2     forall x "model" "model_trace:X":int.
3       (x >= 2) →
4         (forall x1 "model" "model_trace:X@loop":int.
5           (x1 >= 0) →
6            x1 > 0 →
7             (forall x2 "model" "model_trace:X@call":int.
8               (x2 = (x1 - 2)) →
9                ("expl:loop invariant preservation" x2 >= 0)))
```

Figure 9: Logical formula generated by WP.

4.3.2 Instrumenting WP Calculus for Counterexamples

The user expects that all successive values of a variable, marked with label `"model"`, appear in a counterexample. WP creates a fresh logical variable for every modification of a given variable. For variables that are marked with the label `"model"`, counterexample labels are propagated to these fresh logical variables. Moreover, each of these fresh variables is given the location of the expression that triggers its creation.

As an example, see function `test_loop` in Figure 8. The variable `x` is marked with labels `"model"` and `"model_trace"` as counterexample variable. The property of preservation of the loop invariant is not proved and a counterexample, showed in comments, is generated. The formula encoding this property is shown in Figure 9. The variable `x` quantified at the top of the formula stands for the input value of the variable `x` of the `test_loop` function. Then, WP creates another fresh variable `x1` for the value of variable `x` at the beginning of some arbitrary loop iteration. Finally, WP creates a fresh variable `x2` for the value of variable `x` after the assignment statement. As shown on Figure 9, the `"model"` label on the variable `x` of `test_loop` is propagated to all those logical fresh variables corresponding to `x`. A similar propagation occurs with label `"model_trace"` with some additional information (after the `"@"` sign) to explain the origin of the fresh variable. Source code locations are not displayed here for readability.

4.3.3 Get Values of Variables from a Given Assertion

In practice, it is useful for the user to see values of counterexample variables at the location of the assertion that fails. As an example, see Figure 3. Both initial and final value of variable `Val` are displayed on line 6, which is the location of the failed postcondition and this information is also a part of the message summarizing the unproved assertion at the Locations view.

```
1  module M
2    use import ref.Ref
3    use import int.Int
4
5    let test_post (x "model" "model_trace:X" : int)
6                  (y "model" "model_trace:Y" : ref int): unit
7                                        (* counterexample: X = 1 *)
8      ensures { "model_vc" !y >= x }    (* counterexample: Y = 0 ∧ X = 1 *)
9        =
10       y := x - 1                      (* counterexample: Y = 0 *)
11 end
```

Figure 10: Example of WhyML program demonstrating how variables at locations of failed proof are handled. A counterexample is shown in comments.

During WP, all modifications of counterexample variables are marked as a part of the counterexample and their values are displayed at locations of modifications. However, there can be counterexample variables which are just read at the location of the failed proof so displaying values of variables just at the locations of their modifications is not enough. One way to display counterexample variables at the location of a failed proof that are not modified at this location is to retrieve the last point of modification of a variable and displaying the counterexample value at this point. However, this is quite complex to do when multiple program paths are encoded in a VC. That is why we have preferred to explicitly mark variables that appear at the location of a failed proof.

In WhyML programs, expressions that trigger generating a proof task—expressions at locations of potentially failed proofs—are marked with label `"model_vc"`. These expressions can be asserts, preconditions, and postconditions. We wrote a dedicated Why3 transformation that uses this label to find the expression that triggers generating the current proof task. The transformation then marks all counterexample variables read in this expression as a part of the counterexample at locations of the expression.

As an example, see the function `test_post` in Figure 10. It has a postcondition marked with a label `"model_vc"`. This postcondition cannot be proved and a counterexample is generated. At the location of the postcondition, counterexample elements for values of the variable `x` at the function start and the variable `y` on line 10 are displayed.

4.3.4 Projections in Models

For some types, SMT-LIB standard does not specify how values of these types should be displayed. Most notably, these are abstract types. When SMT-LIB solvers are queried for values of such type, they usually return just an internal reference, which does not capture the actual value. To display values of these types in a counterexample, these values must be projected to values of types that can be displayed.

Querying for a value of given Why3 element `E` of a type `T1` that is projected to a value of a different type `T2` is enabled by labeling the element `E` by label `model_projected` and marking a function `P1` that takes a value of the type `T1` as an argument and returns a value of the type `T2` as a projection function with a meta `model_projection`. Instead of querying for a value of the element `E`, a solver will be queried for a value of `P1(E)`. Projections are applied transitively. That is, if there would be a projection function `P2` from `T2` to `T3`, a value of `P2(P1(E))` would be queried.

Projecting values is implemented as a Why3 transformation `intro_projections_cntexmp`. For each value `V` of type `T` marked with a label `model_projected`, the transformation finds the longest sequence of projection functions $P_1, ..., P_n$ where $\forall i, j, 1 \leq i < j \leq n : P_i \neq P_j$, $dom(P_1) = T$, and $\forall i, 1 \leq i < n : range(P_i) = dom(P_{i+1})$. The transformation then introduces new variable VP, marks this variable with labels `model` and `model_trace:content` where `content` is concatenation of the content of

```
1  module Short_int
2    use import "int".Int
3
4    type t
5    function to_rep t : int
6    function of_rep int : t
7    val of_rep (x : int) : t
8      requires { true }
9      ensures { to_rep (result) = x }
10   predicate in_range (x : int)  = ( (-128 <= x) /\ (x <= 128) )
11   axiom range_axiom : forall x : t [to_rep x]. in_range (to_rep x)
12   axiom coerce_axiom : forall x : int [to_rep (of_rep x)].
13       in_range x -> to_rep (of_rep x) = x
14   meta "model_projection" function to_rep
15 end
16
17 module M
18   use Short_int
19
20   constant a "model_projected" "model_trace:a" : Short_int.t
21 end
```

Figure 11: Example of WhyML program demonstrating how abstract types are handled in counterexamples.

`model_trace` label of value `V` with contents of `model_trace` labels of projection functions and introduces and axiom $VP = P_n(P_{n-1}(...P_1(V)))$.

Abstract types As an example, see the definition of an abstract type `Short_Int.t` in Figure 11. This type represents integer values in the range from `-128` to `128`. Values of this type can be projected to integers using function `Short_Int.to_rep`. This function is marked as a projection using meta `model_projection`. The variable `a` of type `Short_Int.t` is marked with the label `model_projection`. This means that the result of the function will be queried in the counterexample and it will be projected using the function `Short_Int.to_rep` from `Short_Int.to_rep` to `int`.

Records To query values of a record types we use the same mechanism of projections. To display values of record types, these values are projected to their fields. For each record field, a function that takes a value of the enclosing record type as a parameter and returns a value of the field is defined and marked as projection function using meta `model_projection`. The function is annotated with a `model_trace` label specifying the name of the field. When the transformation `intro_projections_cntexmp` uses this function to project a record value to the record field, it adds the name of the field to the content of `model_trace` label of the record value. Note that projections are applied transitively—if a record field is of a type with defined projection, it is further projected.

Figure 12 shows an example of definition of record type `r` with fields `f` and `g`. Functions `projf_r_f` and `projf_r_g` project a value of type `r` to field `f` and `g` and they are annotated with `model_trace` labels capturing the names of the fields that will be displayed in a counterexample. The function `b` is marked to be queried for a counterexample with `model_projected` label meaning that the value must be projected before being displayed and it is annotated with `model_trace` label that captures the name of the name of the function that will be displayed in a counterexample.

```
1  module M
2    use import int.Int
3    use Short_int
4
5    type r = {f : Short_int.t; g : bool}
6
7    function projf_r_f "model_trace:.f" (x : r) : Short_int.t = x.f
8    meta "model_projection" function projf_r_f
9
10   function projf_r_g "model_trace:.g" (x : r) : bool = x.g
11   meta "model_projection" function projf_r_g
12
13   constant b "model_projected" "model_trace:b" : r
14 end
```

Figure 12: Example of WhyML program demonstrating how records are handled in counterexamples.

4.3.5 Arrays

SMT-LIB does not define how values of array types should be output in a counterexample. To get values of array types, we rely on the form in which values of array types are returned by the CVC4 solver. CVC4 returns an array as a constant array and series of store operations defining relevant indexes. Here are two examples of array values that CVC4 may return:

```
(store (store ((as const (Array Int Int)) 0) 1 2) 3 4)
((as const (Array Int (Array Int Int))) ((as const (Array Int Int)) 0))
```

The first array is a single-dimensional array with index 1 equal to 2, index 3 equal to 4, and other indexes equal to 0. The second array is a two-dimensional array with all indexes equal to 0.

The values stored in the array may be of abstract or record types so we need to project them. The problem is that we cannot proceed as we did for records by introducing projections for each array index because there are infinitely many of them. To overcome this problem, for an array `orig_arr` that should be queried for a counterexample and has values of an abstract type `t_val`, a projection function `pf_val` from the abstract type of array indexes `t_val` to concrete type `t_val_c` is computed. Then, new array `proj_arr` with values of the type `t_val_c` is defined together with an axiom stating that projections of values in the original array are equal to the values in the new array:

```
constant proj_arr: map int t_val_c
axiom proj_axiom : (forall i : int. proj_arr[i] = pf_val(orig_arr[i]))
```

Instead of querying the solver for the original array, the solver is queried for the new, projected array.

4.4 Building Counterexamples for SPARK

As described in Figure 1, a SPARK program is first translated into a WhyML program. During this translation, counterexample annotations are generated. Why3 then generates verification conditions (proof tasks) and tries to prove proof tasks with selected provers. When non of selected provers is able to proof a given proof task, the task is split to smaller tasks. When a task cannot be neither proved nor split, it is attempted to be proved in a counterexample mode described in Section 4.3. This proof attempt is expected to fail and generate counterexample[3]. The generated counterexample is returned back to Gnat2Why and post-processed. Finally, the counterexample is displayed to the user.

[3]If this proof attempt does not fail, no counterexample is displayed, but the proof task is still considered as unprovable. TODO: explain why? See discussion on ticket OC02-013

4.4.1 Generating WhyML Code

Gnat2Why marks all WhyML elements corresponding to declarations of SPARK variables or to declarations of arguments of SPARK functions to be part of a counterexample using `model` or `model_projected` labels, generates projection functions abstract types generated by Gnat2Why including record types and marks WhyML elements that trigger generating of verification condition by `model_vc` or `model_vc_post` labels. Gnat2Why also generates `model_trace` labels storing traceability information to corresponding elements in SPARK program. Instead of storing names of SPARK elements, `model_trace` labels store identifiers of elements in SPARK AST. This makes it possible to use information from SPARK AST to post-process the counterexample. Finally, Gnat2Why generates Why3 location tags, which make it possible to explicitly specify source code locations of WhyML elements. This way, WhyML program elements have associated locations of corresponding elements in SPARK program.

4.4.2 Post-processing Counterexamples

The counterexample returned from Why3 to Gnat2Why is a map from locations in SPARK source code to list of counterexample elements at these locations. A counterexample element consists of an identifier of the element, information whether the element is an initial value of a function argument, and counterexample value of the element. An identifier of a counterexample element is of the following form:

```
Ident : | Var_Ident
                | Var_Ident "." Record_Fields
Record_Fields : | Var_Ident "." Record_Fields
                | Var_Ident
Variable_Ident : | AST_IDENT Attributes
Attributes : | Attribute Attributes
                | <empty>
Attribute : | "'First"
                | "'Last"
```

An identifier is either an identifier of a simple variable or an identifier of variable followed by identifiers of record fields. Identifier of variable and identifier of record field consists of identifier of a node in an AST of the original SPARK program and optional attribute. Counterexample elements are post-processed in the following way:

- Identifiers of counterexample elements are transformed to SPARK names of counterexample elements by replacing AST identifiers by names of SPARK source code elements that they represent.
- If a counterexample element corresponds to an initial value of a function argument, the AST identifier of the argument is used to get information whether the argument has an `out` modifier in the SPARK program. If the argument has an `out` modifier, an `'Old` attribute is added after the name of the argument.
- If a counterexample element corresponds to declaration of a variable, the AST identifier of the variable is used to find an initializing expression of the variable. If the variable has no initializing expression, the counterexample element is not displayed.
- Counterexample elements in the same SPARK source code line corresponding to variables of record types are grouped together and presented as SPARK record aggregates as shown in Figure 5. Identifiers of AST nodes are used to get information about the record type and this information is used to present record fields in the order of their declaration.
- Values of counterexample elements are converted to SPARK syntax. This includes converting values of enumeration types that are represented as integers in a WhyML program.

5 Conclusions and Perspectives

We added a counterexample feature to SPARK, by exploiting the model generation feature of SMT solvers, and appropriately instrumenting the process of generating verification conditions from a SPARK program, through an intermediate WhyML code, weakest precondition calculus and logic transformations. Instead of a probably complex post-processing of the complete model that would be returned by the SMT-LIB `get-model` command, we instrumented the processing steps so that only the adequate terms are queried with the `get-value` command, and then a simple mapping from the term queried to the initial program variable can be applied to build the counterexample.

Recent user training sessions showed a clear appeal of counterexamples on users, which motivated our choice to enable them by default in SPARK. The counterexample feature is now distributed since the version SPARK Pro 16.0. Based on our initial feedback with the use of counterexamples inside SPARK, counterexamples may be the most useful feature in SPARK for investigating unproved properties, after the possibility to execute contracts and assertions in tests.

5.1 Related Work

The model returned by an SMT solver on a satisfiable problem is exploited in several areas of program verification, a major case being the one of model checking, as for example in the CBMC model checker for C programs [15].

In the case of deductive verification, the techniques for generating counterexamples are not so many. There is the case of the NitPick tool inside the Isabelle proof assistant [4] which uses a technique that is different from ours, not making use of an SMT solver. In the more specific case of program verifiers using SMT solvers, the idea of instrumenting the generation of verification conditions originates from the old system ESC/Modula-3, that generates VCs for the Simplify solver, adding specific labels to determine the source location of the potential program error. The same mechanism was reused in ESC/Java. The potential counterexample proposed by Simplify can be displayed to the user, but is very hard to understand because of the various encodings from the input program to the VC. Only recently a way to reinterpret the counterexample in terms of variables of the source code was designed in the OpenJML framework [11]. Another deductive program verification framework that makes use of SMT counter-models is the Boogie Verifier Debugger [17]. Boogie is used as an intermediate language by Dafny [18] and VCC [10]. Boogie also has its own way of reinterpreting the counter-model, generated by its back-end prover Z3, in terms of the source code. Both OpenJML and Boogie present the counterexample in a user-friendly manner, in their respective graphical interfaces (Eclipse, Visual Studio). Their presentation is a bit different from our way of presenting the counterexample, where we give values of relevant variables inside comments at proper locations of the source code. We do not know if OpenJML and Boogie internally use the SMT-LIB `get-value` command as we do, or if they proceed differently.

Another recent approach for helping users in debugging their specification and code is to use some kind of symbolic execution, as is proposed by the Visual Studio dynamic debugger [20] and the Verifast verifier [16].

5.2 Future Work

Even if the support for counterexamples is already quite satisfactory in the SPARK environment, it should be noted that we hit a few issues, in particular issues that could be addressed by authors of SMT solvers.

First, even if the `get-model` and `get-value` commands are part of SMT-LIB standard, there are no fixed rules for displaying counterexample values yet. In particular, it is not standardized how values of array types, record types, and bitvector types should be displayed. The need for standardization of model display is already known and it is likely to appear in a near future. Related to this, it seems that the

feature of projections, that we introduced to handle abstract types, could be taken care of by the solvers themselves as part of the standard to display counterexamples. This would particularly useful in the case of arrays: the solution we proposed, involving the introduction of another array and an axiom, makes the problem harder to prove because of the additional universal quantification.

A second issue concerns the validity of counterexamples generated. In principle, one should query SMT solvers for models only if the answer was `'sat'`. However, on a VC coming from a program verification task, the answer is usually `'unknown'` or, in fact in the majority of cases, the solver hits the time limit given. In such a case, the model is of course not guaranteed to be a true model. However, there are some cases where the model returned is trivially wrong because it is not even a model of the ground part of the goal. Thus there seems to be some room for improvement. A suggestion is as follows: since the main source of incompleteness comes from the quantified hypotheses, there could be two different modes of operation, with two corresponding time limits. A first time limit, say a "soft" one, gives the time in which the solver is allowed to instantiate quantifiers as it wants. After this soft time limit is reached, a "hard" time limit should give the solver extra time to continue its search but in a specific mode where no new quantifier instantiation is performed. In this second mode, it is likely that the solver would terminate its search, and if a model is returned, it would be valid with respect to the ground part of the goal. If such modes were implemented in SMT solvers, it would be of major interest for counterexample generation.

Another technical issue is the ability to support model generation for all supported theories. This is not always the case, for example CVC4 does not produce models when the NIA logic (non-linear arithmetic) is selected. It is understandable since this logic is undecidable, there is no way to be sure that the model returned would be a true one. However, a similar degraded mode as described above could be implemented, for example in the degraded mode non-linear parts of the formulas could be ignored.

A last issue regarding the models generated by SMT solvers concerns a question of minimality. A model always contains concrete values for all the variables queried in the `get-value` command. However, especially when there are a lot of variables, it is often the case that for some variables any of their values will be convenient. In such a case, an equivalent of "any" value could be returned. The counterexample presented to the user would then not show any value for this particular variable.

More generally, to double-check that a counterexample produced by our technique is a true one or not, one may think of turning it into a test case and run the program with the given values. This is unfortunately not an easy task because of the procedure calls: program verification considers procedures called as black-boxes with effect only visible through their contracts. This is not the case for a concrete execution, and there might exist some data, not visible from the current procedure (e.g. local data of other packages) that would need some values for the program to run. Thus, combining properly counterexamples generated by failed proof attempts and runtime verification needs to investigated further. A very recent work by Christakis et al. [9] goes into such a direction.

References

[1] John Barnes. *Programming in Ada 2012*. Cambridge University Press, 2014.

[2] 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.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.

[4] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[5] 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/`.

[6] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[7] 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`.

[8] Roderick Chapman and Florian Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2014.

[9] 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)*. Springer, 2016.

[10] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[11] 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.

[12] 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.

[13] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[14] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[15] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 453–456. Springer, 2004.

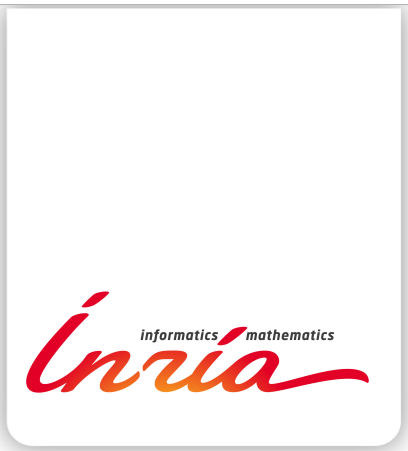
[16] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[17] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The boogie verification debugger. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods - 9th International Conference, (SEFM)*, volume 7041 of *Lecture Notes in Computer Science*, pages 407–414. Springer, 2011.

[18] 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.

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

[20] Peter Müller and Joseph N. Ruskiewicz. Using debuggers to understand failed verification attempts. In Michael J. Butler and Wolfram Schulte, editors, *17th International Symposium on Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2011.



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