

$2 * 2 \neq 12$ 166 397

Trusted Counterexamples in SPARK

Benedikt Becker, Claude Marché, Yannick Moy

AdaCore Paris

October 21, 2021

Examples for proof failures

```
procedure Example1 (X : Natural)
is
  Y : Natural := X + 1;
begin
  pragma Assert (Y /= 43); ⚡
end Example1;
```

- ▷ `gnatprove:medium:` assertion might fail (e.g. when $Y = 43$)
- ▷ proof failure due to **non-conformance** between code and assertion

Examples for proof failures

```
procedure Incr (X: in out Natural)
  with Post => (X > X'Old) is
begin
  X := X + 1;
end Incr;
```

```
procedure Example2 (Y : in out Natural)
  with Post => (Y = Y'Old + 1) ⚡
is
begin
  Incr (Y);
end Example2;
```

- ▷ gnatprove: medium: postcondition might fail (e.g. when $Y'Old = 0$ and $Y = 2$)
- ▷ proof failure due to a **subcontract-weakness**

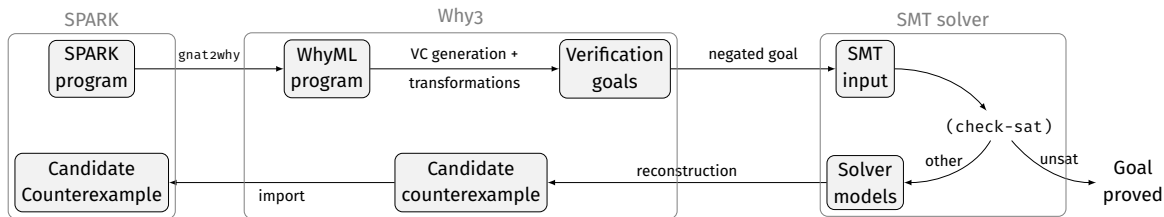
Examples for proof failures

```
procedure Example3 (A, B: in Natural)
  with Pre => A >= 2 and B >= 2
is
  C : constant Natural := 12166397;
begin
  pragma Assert (A * B /= C); ⚡
end Example3;
```

- ▷ `gnatprove:medium:` assertion might fail (e.g. when $A = 2$ and $B = 2$)
- ▷ **bad counterexample!**

Candidate counterexample generation in Why3

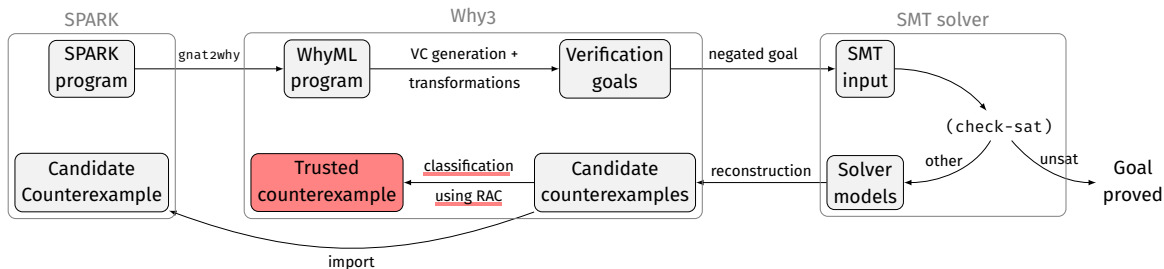
Dailler, Hauzar, Marché, Moy (2018): *Instrumenting a Weakest Precondition Calculus for Counterexample Generation*



- ▷ no guarantee on the validity of the solver models
→ potentially **bad counterexamples**
- ▷ **no hints on the reason** of the proof failure

I. Trusted counterexamples in Why3

Becker, Belo Lourenço, Marché (2021): *Explaining Counterexamples with Giant-Step Assertion Checking*



Motivation: make counterexamples more helpful for users

- ▷ validate candidate counterexamples
- ▷ categorise proof failures as non-conformity or subcontract weakness using normal + *giant-step* runtime assertion checking

Outline

I. Trusted counterexamples in Why3

- 1 Runtime assertion checking in Why3
- 2 *Giant-step* runtime assertion checking
- 3 Validation of counterexamples and categorisation of proof failures

II. Trusted counterexamples in SPARK

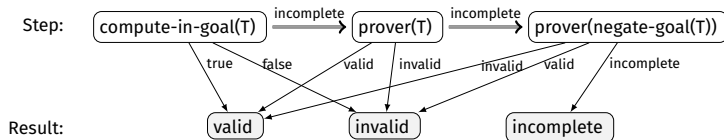
Normal runtime assertion checking in Why3

- ▷ normal program execution, validity of annotations are checked
- ▷ **invalid annotations terminate execution**
 - ▷ Failure for assertions
 - ▷ Stuck for assumptions

Normal runtime assertion checking in Why3

- ▷ normal program execution, validity of annotations are checked
- ▷ **invalid annotations terminate execution**
 - ▷ Failure for assertions
 - ▷ Stuck for assumptions
- ▷ Why3's annotation language is **not executable**

1. steps to check an annotation



2. "incomplete" may or may not terminate execution (configurable)
3. checked annotations as preconditions for subsequent checks

Runtime assertion checking of a counterexample

```
let example1 (x: int)
= let y = x + 1 in
  assert { y <> 43 } ⚡
```

▷ counterexample: x=42

Preparation

1. find program function from where the verification goal originates
2. initialise arguments for initial function call and global variables with values from counterexample

Runtime assertion checking of a counterexample

```
let example1 (x: int)
= let y = x + 1 in
  assert { y <> 43 } ⚡
```

▷ counterexample: $x=42$

▷ normal RAC: `main 42` \rightsquigarrow Failure

Preparation

1. find program function from where the verification goal originates
2. initialise arguments for initial function call and global variables with values from counterexample

Intermediate result

▷ failure in normal RAC \Rightarrow non-conformity between program and annotation

But how to identify a sub-contract weakness?

Giant-step runtime assertion checking

Deductive program verification is modular

- ▷ from the outside, function and loops are defined by their post-condition and invariants (**sub-contracts**), not their bodies
- ▷ counterexamples values for function calls and loops comply to the sub-contracts (usually!)

Giant-step runtime assertion checking

Deductive program verification is modular

- ▷ from the outside, function and loops are defined by their post-condition and invariants (**sub-contracts**), not their bodies
- ▷ counterexamples values for function calls and loops comply to the sub-contracts (usually!)

Idea of giant-step RAC: like normal RAC but

- ▷ don't execute function bodies, don't iterate loop bodies
- ▷ **retrieve return values and values of written variables from oracle**

Giant-step runtime assertion checking

Function calls

RAC execution of a function call

$f v_1 \cdots v_n$

at location p in environment Γ , with

let $f x_1 \dots x_n$ **writes** $\{ y_1, \dots, y_m \}$
requires $\{ \phi_{\text{pre}} \}$ **ensures** $\{ \phi_{\text{post}} \} = e$

1. bind arguments to parameters
2. assert pre-conditions
3. **normal RAC:**
evaluate body e to result value v , modifying written variables by side-effect
4. assert post-conditions
5. return value v

$\Gamma_1 := \Gamma[\dots, x_i \leftarrow v_i, \dots]$

$\Gamma_1 \vdash \phi_{\text{pre}}$

$(v, \Gamma_2) := \text{eval}(e, \Gamma_1)$

$\Gamma_2[\text{result} \leftarrow v] \vdash \phi_{\text{post}}$

(v, Γ_2)

Giant-step runtime assertion checking

Function calls

RAC execution of a function call

$f v_1 \cdots v_n$

at location p in environment Γ and **oracle** Ω , with

let $f x_1 \dots x_n$ **writes** $\{ y_1, \dots, y_m \}$
requires $\{ \phi_{\text{pre}} \}$ **ensures** $\{ \phi_{\text{post}} \} = e$

1. bind arguments to parameters
2. assert pre-conditions
3. **giant-step RAC**:
retrieve result value v and
update written variables from oracle
4. **assume** post-conditions
5. return value v

$\Gamma_1 := \Gamma[\dots, x_i \leftarrow v_i, \dots]$

$\Gamma_1 \vdash \phi_{\text{pre}}$

$v = \Omega(\text{result}, p)$

$\Gamma_2 := \Gamma_1[\dots, y_i \leftarrow \Omega(y_i, p), \dots]$

$\Gamma_2[\text{result} \leftarrow v] \vdash \phi_{\text{post}}$

(v, Γ_2)



Giant-step runtime assertion checking

While loops

RAC execution of a while loop at location p
in environment Γ and **oracle** Ω :

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


1. assert invariant (initialisation)
2. **giant-step:**
 - ▷ update written variables from oracle
 - ▷ assume invariant
3. if condition e_1 is true
 - ▷ evaluate loop body e_2
 - ▷ assert invariant (preservation)
 - ▷ **stuck**
4. else done


$$\Gamma \vdash \phi_{inv}$$
$$\Gamma_1 := \Gamma[\dots, y_i \leftarrow \Omega(y_i, p), \dots]$$
$$\Gamma_1 \vdash \phi_{inv}$$
$$(true, \Gamma_2) := eval(e_1, \Gamma_1)$$
$$((), \Gamma_3) := eval(e_2, \Gamma_2)$$
$$\Gamma_3 \vdash \phi_{inv}$$
$$((), \Gamma_2)$$

Identification of a sub-contract weakness

Giant-step RAC of a counterexample

```
let incr (x: int) : int
  ensures { result > x }
= x + 1
```


```
let example2 (x: int)
= let y = incr x in
  assert { y = x + 1 } 
```

- ▷ counterexample: $x=0, y=2$
- ▷ find program function from where the verification goal originates
- ▷ two executions: normal RAC and giant-step RAC
- ▷ counterexample as oracle for
 - ▷ initial values of global variables + arguments for initial function call
 - ▷ written variables and return values in giant-step RAC

Identification of a sub-contract weakness

Giant-step RAC of a counterexample

```
let incr (x: int) : int
  ensures { result > x }
= x + 1
```

```
let example2 (x: int)
= let y = incr x in
  assert { y = x + 1 } 
```

▷ counterexample: $x=0, y=2$

▷ normal RAC: $\text{example2 } 0 \rightsquigarrow$ **Normal termination**

- ▷ find program function from where the verification goal originates
- ▷ two executions: normal RAC and giant-step RAC
- ▷ counterexample as oracle for
 - ▷ initial values of global variables + arguments for initial function call
 - ▷ written variables and return values in giant-step RAC

Identification of a sub-contract weakness

Giant-step RAC of a counterexample

```
let incr (x: int) : int
  ensures { result > x }
= x + 1
```

```
let example2 (x: int)
= let y = incr x in
  assert { y = x + 1 } ⚡
```

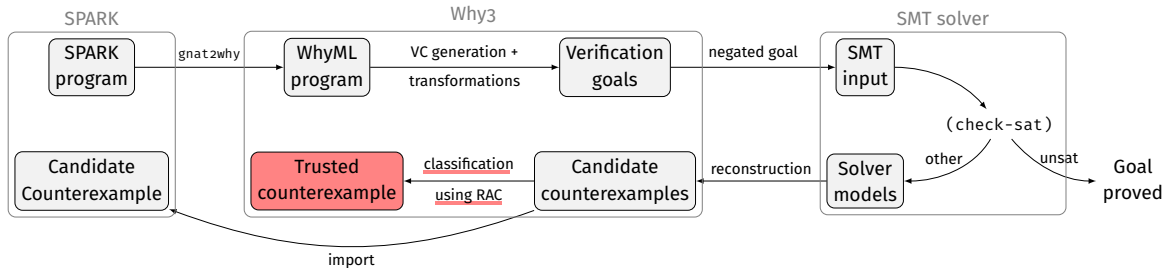
- ▷ counterexample: $x=0, y=2$
 - ▷ normal RAC: $\text{example2 } 0 \rightsquigarrow$ Normal termination
 - ▷ giant-step RAC: $\text{example2 } 0, \text{incr } x = 2 \rightsquigarrow$ Failure
- ▷ find program function from where the verification goal originates
- ▷ two executions: normal RAC and giant-step RAC
- ▷ counterexample as oracle for
 - ▷ initial values of global variables + arguments for initial function call
 - ▷ written variables and return values in giant-step RAC

Classification of candidate counterexamples (CE)

Normal RAC	Giant-step RAC			
	Failure	Normal	Incomplete	Stuck
Failure	Non-conformity if failure matches goal else Bad CE			
Normal	Sub-contract weakness	Bad CE	Incomplete	Bad CE
Incomplete	Non-conformity or sub-contract weakness	Incomplete	Incomplete	Bad CE
Stuck	Bad CE (Invalid assumption)			

I. Trusted counterexamples in Why3

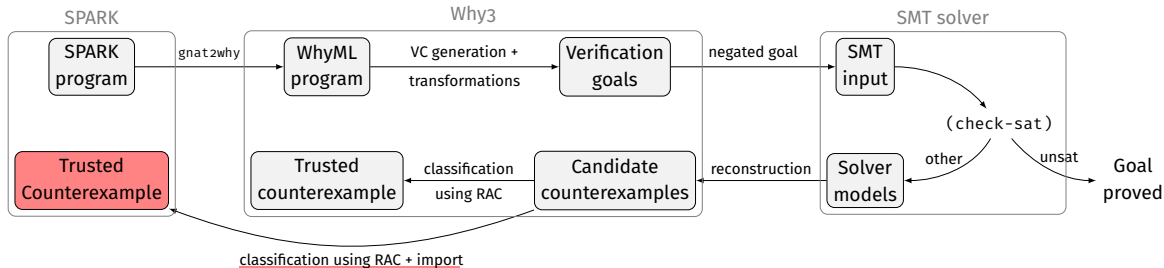
Becker, Belo Lourenço, Marché (2021): *Explaining Counterexamples with Giant-Step Assertion Checking*



Motivation: make counterexamples more helpful for users

- ▷ validate candidate counterexamples
- ▷ categorise proof failures as non-conformity or subcontract weakness using normal + *giant-step* runtime assertion checking

II. Trusted counterexamples in SPARK



- ▷ generated WhyML program not executable
- ▷ classification in `gnat2why` based on results from
 - ▷ giant-step RAC of generated program in Why3
 - ▷ normal RAC of original program in SPARK

Concrete RAC in SPARK

- ▷ implemented in Ada as part of `gnat2why`
- ▷ annotations are computed
- ▷ limited by stack height and “fuel”
- ▷ result is combined with result of giant-step RAC from `Why3`

Results

```
procedure Example1 (X : Natural)
is
  Y : Natural := X + 1;
begin
  pragma Assert (Y /= 43); ⚡
end Example1;
```

- ▷ high: assertion might fail
(e.g. when $Y = 43$)

```
procedure Example3 (A, B: in Natural)
  with Pre => A >= 2 and B >= 2
is
  C : constant Natural := 12166397;
begin
  pragma Assert (A * B /= C); ⚡
end Example3;
```

- ▷ medium: assertion might fail

```
procedure Incr (X: in out Natural)
  with Post => (X > X'Old) is
begin
  X := X + 1;
end Incr;
```

```
procedure Example2 (Y : in out Natural)
  with Post => (Y = Y'Old + 1) ⚡
is
begin
  Incr (Y);
end Example2;
```

- ▷ medium: postcondition might fail
(e.g. when $Y'Old = 0$ and $Y = 2$)
[tip: add or complete related loop invariants or postconditions]

Current state

Statistics on the SPARK testsuite with 3 391 counterexamples in 39150 checks

Result	Count	Percentage
Incomplete checking	<u>1 614</u>	(47.58%)
Checked counterexamples	1 778	(52.42%)
– bad counterexample	621	– (34.93%)
– non conformity	617	– (34.70%)
– non conformity or subcontract weakness	<u>464</u>	– (26.10%)
– subcontract weakness	76	– (4.27%)

Future work

- ▷ add support for more SPARK language features
- ▷ more return values and values of written variables in candidate counterexample
- ▷ identifying single sub-contract weaknesses
- ▷ dealing with incomplete oracles

Code dive!

Ideas

- ▷ classification table in `gnat2why`
- ▷ locations and position attributes for return values in generated WhyML program
- ▷ giant-step RAC in `gnatwhy3`
- ▷ CE classification in Why3
- ▷ ...