SMT solvers in Deductive Program Verification

Claude Marché

Inria team Toccata

Journée SMT, March 29th, 2016
Outline

1. Introduction to Why3 and front-ends

2. Theories Needed in Deductive Verification
   - Integer Arithmetic
   - Functional Arrays
   - Quantifiers
   - Datatypes
   - Bitvectors
   - Reals and Floats

3. Generation of Counterexamples

4. Conclusions
1 Introduction to Why3 and front-ends

2 Theories Needed in Deductive Verification
   Integer Arithmetic
   Functional Arrays
   Quantifiers
   Datatypes
   Bitvectors
   Reals and Floats

3 Generation of Counterexamples

4 Conclusions
Why3 in a nutshell

Why3 core:

- rich specification language: *extended first-order logic*
  - arithmetic, type polymorphism, algebraic data types, inductive predicates, higher-order functions, etc.
- architecture to *send proof goals to external provers*
  - complex constructs encoded into simpler logic
  - printers producing adequate syntax for SMT solvers, interactive proof assistants, etc.

Why3 programming language "WhyML"

- Functional, ML-like syntax
- Mutable data types, with statically controlled aliasing
- Contracts, VC generator
My First Why3 example

Demo
Multiplication in binary

let binary_mult (a b: int)
  ensures { result = a * b }
= let x = ref a in
  let y = ref b in
  let z = ref 0 in
  while !y ≠ 0 do
    if !y % 2 ≠ 0 then z := !z + !x;
    x := !x + !x;
    y := !y / 2
  done;
  !z
Why3 ecosystem

Java programs
- Krakatoa

C programs
- Frama-C

Ada programs
- SPARK2014

SMT solvers
- Alt-Ergo
- CVC4
- veriT
- Z3
- etc.

Interactive provers
- Coq
- Isabelle
- PVS

Other provers
- E prover
- Gappa
- SPASS
- Vampire
- etc.
Why3 ecosystem

Java programs
- Krakatoa

C programs
- Frama-C

Ada programs
- SPARK2014

Why3 ecosystem

SMT solvers
- Alt-Ergo
- CVC4
- veriT
- Z3
- etc.

Interactive provers
- Coq
- Isabelle
- PVS

Other provers
- E prover
- Gappa
- SPASS
- Vampire
- etc.
Why3 ecosystem

Java programs
- Krakatoa

C programs
- Frama-C

Ada programs
- SPARK2014

SMT solvers
- Alt-Ergo
- CVC4
- veriT
- Z3
- etc.

Interactive provers
- Coq
- Isabelle
- PVS

Other provers
- E prover
- Gappa
- SPASS
- Vampire
- etc.
Why3 ecosystem

- Java programs: Krakatoa
- C programs: Frama-C
- Ada programs: SPARK2014

**SMT solvers**
- Alt-Ergo
- CVC4
- veriT
- Z3
- etc.

**Interactive provers**
- Coq
- Isabelle
- PVS

**Other provers**
- E prover
- Gappa
- SPASS
- Vampire
- etc.
Why3 ecosystem

Java programs
- Krakatoa

C programs
- Frama-C

Ada programs
- SPARK2014

SMT solvers
- Alt-Ergo
- CVC4
- veriT
- Z3
- etc.

Interactive provers
- Coq
- Isabelle
- PVS

Other provers
- E prover
- Gappa
- SPASS
- Vampire
- etc.
How Why3 uses SMT solvers?

*Verification condition* for a given procedure: computed using a *Weakest Precondition* calculus

\[
WP(e, Q)
\]

property: in any state satisfying formula \( WP(e, Q) \):

- *e executes safely*
- If it terminates, \( Q \) is valid in the final state
- Or ("strict" variant) it terminates and \( Q \) is valid
Weakest Precondition Calculus

- Pure expressions:
  \[ WP(t, Q) = Q[result \leftarrow t] \]

- Sequence:
  \[ WP((e_1; e_2), Q) = WP(e_1, WP(e_2, Q)) \]

- Assignment:
  \[ WP(x := t, Q) = Q[x \leftarrow t] \]

- Conditional
  \[ WP(if t then e_1 else e_2, Q) = \\
     if t then WP(e_1, Q) else WP(e_2, Q) \]
Weakest Precondition Calculus

- **Assertion**
  \[
  WP(\text{assert } P, Q) = P \land (P \rightarrow Q)
  \]

- **While loop**
  \[
  WP(\text{while } t \text{ invariant } l \text{ do } e, Q) = \\
  l \land \forall \vec{w}, (l \rightarrow \text{if } t \text{ then WP}(e, l) \text{ else } Q)
  \]

- **Procedure call**
  \[
  WP(f(t_1, \ldots, t_n), Q) = \\
  \text{Pre}[x_i \leftarrow t_i] \land \forall \vec{w}, (\text{Post}[x_i \leftarrow t_i] \rightarrow Q)
  \]
Using a SMT solver to discharge a VC

- VC for a procedure: $\text{Pre} \rightarrow \text{WP}(\text{Body}, \text{Post})$
  - Boolean combination of user annotations
  - only “prenex” universal quantification

- Calling SMT solver:
  - goal is *negated*, asking solver for unsatisfiability
  - Given the shape of WP calculus, this results in a set of propositional clauses

- All the complexity comes from:
  - the *user annotations*
  - the context:
    - *theories* that the program may use
    - *axiomatization of program logic* in case of C/Java/Ada programs
Related Similar Tools

Historically: ESC-Modula, ESC-Java
  • First SMT solver *Simplify* [Nelson, 1980]

Boogie and front-ends:
  • Spec#, VCC, Dafny,
  • mainly use Z3

JML tools
  • OpenJML
  • use various provers

B tools
  • WP $\rightarrow$ refinement calculus
  • home made provers
  • recently: use of veriT, Alt-Ergo, etc.
Outline

1. Introduction to Why3 and front-ends

2. Theories Needed in Deductive Verification
   - Integer Arithmetic
   - Functional Arrays
   - Quantifiers
   - Datatypes
   - Bitvectors
   - Reals and Floats

3. Generation of Counterexamples

4. Conclusions
Integer Arithmetic

- Unbounded integers, Linear but also non-linear
- Naturally ubiquitous
- First reason why SMT solvers are superior to TPTP provers for deductive verification

Also used to formalize machine integers:

```why3

type uint8                  (* abstract type *)

function to_int uint8 : int

axiom uint8_range:
    forall x:uint8. 0 ≤ to_int x ≤ 255

val add (x y:uint8) : uint8   (* abstract procedure *)
    requires { "expl:integer_overflow"
                to_int x + to_int y ≤ 255 }
    ensures { to_int result = to_int x + to_int y }
```

- Notice the appearance of a quantifier in the context
Integer Arithmetic

- Unbounded integers, Linear but also non-linear
- Naturally ubiquitous
- *First reason why SMT solvers are superior to TPTP provers for deductive verification*

Also used to *formalize machine integers*:

```plaintext
type uint8 (* abstract type *)
function to_int uint8 : int
axiom uint8_range:
    forall x:uint8. 0 ≤ to_int x ≤ 255
val add (x y:uint8) : uint8 (* abstract procedure *)
    requires { "expl:integer_overflow"
        to_int x + to_int y ≤ 255 }
    ensures { to_int result = to_int x + to_int y }
```

- Notice the appearance of a *quantifier in the context*
Functional Arrays

- To formalize programs with arrays
- McCarthy’s theory of functional arrays

```
type map 'a    (* abstract, polymorphic type *)
function select (map 'a) int : 'a
function store (map 'a) int 'a : map 'a

axiom select_store_eq: forall m i v.
   select (store m i v) i = v
axiom select_store_neq: forall m i j v.
   i ≠ j → select (store m i v) j = select m j

val set (m:ref (map 'a)) (len:int) (i:int) : unit
  requires { "expl:index_in_bounds" 0 ≤ i < len }
  writes { m }
  ensures { !m = store (old !m) i v }
```

- SMT solvers support this theory very well
- Examples: searching, sorting, etc.
Example: binary search

```
let binary_search (a : array int) (v : int)
= let m = ref 0 in try
  let l = ref 0 in let u = ref (a.length - 1) in
  while !l <= !u do
    m := !l + (!u - !l) / 2;
    if a[!m] < v then l := !m + 1 else
      if a[!m] > v then u := !m - 1 else raise Break
  done;
  raise Not_found
with Break → !m
```
Example: binary search

```why3
let binary_search (a : array int) (v : int)
  requires { forall i j. 0 ≤ i ≤ j < length a → a[i] ≤ a[j] } 
  ensures { 0 ≤ result < a.length ∧ a[result] = v } 
  raises { Not_found → forall i. 0 ≤ i < length a → a[i] ≠ v }
  = let m = ref 0 in try
    let l = ref 0 in let u = ref (a.length - 1) in
    while !l ≤ !u do
      m := !l + (!u - !l) / 2;
      if a[!m] < v then l := !m + 1 else
        if a[!m] > v then u := !m - 1 else raise Break
    done;
    raise Not_found
  with Break → !m
```

- Notice the appearance of *user quantifiers*
Example: binary search

```
let binary_search (a : array int) (v : int)
  requires { forall i j. 0 ≤ i ≤ j < length a → a[i] ≤ a[j] }
  ensures { 0 ≤ result < a.length ∧ a[result] = v }
  raises { Not_found → forall i. 0 ≤ i < length a → a[i] ≠ v }
= let m = ref 0 in try
  let l = ref 0 in let u = ref (a.length - 1) in
  while !l ≤ !u do
    invariant { 0 ≤ !l ∧ !u < a.length }
    invariant {
      forall i. 0 ≤ i < a.length → a[i] = v → !l ≤ i ≤ !u }
    variant { !u - !l }
    m := !l + (!u - !l) / 2;
    if a[!m] < v then l := !m + 1 else
      if a[!m] > v then u := !m - 1 else raise Break
  done;
  raise Not_found
with Break → !m
```

- Notice the appearance of *user quantifiers*
Quantifiers

Quantifiers are *ubiquitous* in

- *user annotations* (e.g. programs on arrays)
- *libraries of theories* e.g.
  - B set theory
- *formalization of specific program logics* e.g.
  - Models of memory heap for C programs

→ SMT solvers without quantifier support are rarely used in deductive program verification.

Quantifiers raise many issues

- Completeness is lost
- SMT trigger-based instantiation: efficient but not easy to master
- In some cases, TPTP provers are better than SMT solvers
Quantifiers

Quantifiers are *ubiquitous* in

- *user annotations* (e.g. programs on arrays)
- *libraries of theories* e.g.
  - B set theory
- *formalization of specific program logics* e.g.
  - Models of memory heap for C programs

→ SMT solvers without quantifier support are rarely used in deductive program verification.

Quantifiers raise many issues

- Completeness is lost
- SMT trigger-based instantiation: efficient but not easy to master
- In some cases, TPTP provers are better than SMT solvers
Higher-order quantification

There is even need for higher-order quantification
[Clochard et al., VSTTE 2014]

Why3 applies \textit{transformations} to reduce it to first-order

- lambda-lifting

There is no need to support higher-order in the provers
Datatypes

\[
\text{type } \tau = \\
\mid C_1 \tau_{1,1} \cdots \tau_{1,n_1} \\
\mid \vdots \\
\mid C_k \tau_{k,1} \cdots \tau_{k,n_k}
\]

- Supported in SMT-LIB 2.0

Frequent in program verification:
- Particular case of \textit{records}
- Particular case of \textit{enumerated types}
- Standard examples: \textit{lists}, \textit{trees}
- Need for \textit{pattern-matching} (not in SMT-LIB)

Why3 encodes pattern-matching before calling provers
Datatypes and Induction

```
type list 'a = Nil | Cons 'a (list 'a)

function append (l1:list 'a) (l2:list 'a) : list 'a =
  match l1 with
  | Nil → l2
  | Cons x l → Cons x (append l l2)
  end

lemma append_assoc:
  forall l1 l2 l3:list 'a.
    append (append l1 l2) l3 = append l1 (append l2 l3)
```

Such a lemma is not accessible to SMT solvers: needs
induction
Induction by lemma functions

let rec lemma append_assoc (l1 l2 l3:list 'a) : unit
  variant { l1 }
  ensures {
    append (append l1 l2) l3 = append l1 (append l2 l3) }
  = match l1 with
    | Nil → ()
    | Cons _ r → append_assoc r l2 l3
  end

• Program function proved correct and terminating
• Lift into the logic context (keyword lemma)
  • allowed because it is side-effect free
Bitvectors

Fixed-size bitvectors

- Well supported by SMT-LIB
- Useful to model bitwise operators in programs
  - Used to model *Modular types* of Ada in SPARK (since version 16.0, ProofInUse project)

Issues:

- In practice, it is desirable to specify such low-level code with higher-level specifications
- Mixing bitvectors with arithmetic is not easy for solvers
Example: bit counting

```c
//@ ensures to_uint result = numof (\i. nth x i) 0 32;
uint32_t count(uint32_t x) {
    x = x - ((x >> 1) & 0x55555555) ;
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333) ;
    x = (x + (x >> 4)) & 0x0F0F0F0F ;
    x = x + (x >> 8) ;
    x = x + (x >> 16) ;
    return (x & 0x0000003F) ;
}
```

- Proved a Why3 version of this code
  - using a combination of SMT-solvers
- See [Fumex et al., Inria research report 8821]
Reals and Floats

In the context of Frama-C

Specification and proof of *floating-point programs*

- hand-made partial *axiomatization of FP arithmetic*
- on top of *real arithmetic*
- use of the specialized prover *Gappa*

Future Work (ProofInUse project):
- Exploit new support for IEEE-754 FP arithmetic in SMT solvers
  - for Ada programs in particular
Remez interpolation of exponential on $[-1, 1]$

```c
/*@ requires \abs(x) <= 1.0;
   @ ensures \abs(\result - \exp(x)) <= 0x1p-4;
   @*/

double my_exp(double x) {
  return 0.9890365552 + 1.130258690*x + 0.5540440796*x*x;
}
```
Example

```c
/*@ requires \abs(x) <= 1.0; 
@ ensures \abs(result - \exp(x)) <= 0x1p-4; 
@*/
double my_exp(double x) {
  /*@ assert \abs(0.9890365552 + 1.130258690*x + 
  @ 0.5540440796*x*x - \exp(x)) <= 0x0.FFFFp-4; 
  @*/
  return 0.9890365552 + 1.130258690*x + 0.5540440796*x*x;
}
```

Proof done using

- Gappa (absence of overflow + post-condition)
- Coq-interval (the assertion)

See [Boldo & Marché, Mathematics in Comp. Sc., 2011]
1. Introduction to Why3 and front-ends

2. Theories Needed in Deductive Verification
   - Integer Arithmetic
   - Functional Arrays
   - Quantifiers
   - Datatypes
   - Bitvectors
   - Reals and Floats

3. Generation of Counterexamples

4. Conclusions
Need for Counterexamples

When a proof does not succeed, what the user can do?

- *Increase time limit*
- Try *more provers*
- Think hard and *find extra assertions*

But when SMT solver answers \texttt{sat}

- It provides a *model*
- This model can be turned into a (potential) *counterexample*
Illustration in SPARK

```ada
procedure Saturate (Val : in out Unsigned_16) with
    Post =>
    (if Val'Old <= 255 then Val = Val'Old) and
    (if Val'Old >  255 then Val = 255)
is
begin
    Val := Val and 16#FF#;
end Saturate;
```

Messages

Locations

Builder results (1 item)

saturation.adb (1 item)

6:7 medium: postcondition might fail
Illustration in SPARK

```ada
procedure Saturate (Val : in out Unsigned_16)
begin
  with Post =>
  (if Val'Old <= 255 then Val = Val'Old) and
    Val'Old = 4096 and Val = 0
    (if Val'Old > 255 then Val = 255)
  Val := Val and 16#FF#;
  Val = 0
end Saturate;
```

Messages | Locations
--- | ---

Builder results (1 item)
saturation.adb (1 item)

6:7 medium: postcondition might fail (e.g. when
Counterexamples issues

- Counterexamples with non-linear arithmetic?
- If the time limit is reached
  - no counterexample
- If answer is unknown (incompleteness):
  - counterexample may be wrong
- In practice: may even be *trivially wrong*
  - Not a model of the ground part of the problem

Possible solution that SMT solvers could implement:
- Two time limits
- Once first time limit reach, focus on ground part

See [Hauzar *et al.*, Inria research report 8854]
1 Introduction to Why3 and front-ends

2 Theories Needed in Deductive Verification
   Integer Arithmetic
   Functional Arrays
   Quantifiers
   Datatypes
   Bitvectors
   Reals and Floats

3 Generation of Counterexamples

4 Conclusions
Conclusions

For application to deductive program verification:

- SMT theories come to play for *modeling data*
- *Many theories* are useful
  - non-linear integer and real arithmetic
  - arrays, datatypes
  - bitvectors, floating-point numbers
  - first-order *quantifiers*
- Some features can be handled before calling the solver
  - Polymorphic types
  - Pattern-matching
  - Higher-order functions
  - Reasoning by induction
- Often, *specifications are written on a logic larger than the program’s logic*
- Decidability/Completeness results are not mandatory
Some Possible Future Work

- **Counterexamples:**
  - soft versus hard time limits
  - Counterexamples for non-linear arithmetic

- **Alternatives to first-order axiomatization**
  - Quantifiers: alternative to triggers?
  - Define theories using rewrite rules?

- **Combine SMT approach with other Constraint Programming approaches, with resolution/paramodulation, etc.**