Using SMT Solvers for Deductive Verification of C and Java Programs

Jean-Christophe Filliâtre

CNRS
Orsay, France

SMT, July 7, 2008

The ProVal Project — `http://proval.lri.fr/`

- foundations of **ProVal**: the Coq project
  - type theory: type $\simeq$ logic specification
  - Curry-Howard isomorphism: proof $\simeq$ program
  - **functional** programs only
- goals of **ProVal**:
  - to deal with **imperative programs** (C, Java)
  - to apply our methods to **industrial cases**

Brief History

- 1999: a first approach for programs with side effects in Coq
- 2000-2003: EU project Verificard (verification of Java Card applets with industrial partners GemPlus, Schlumberger)
- 2001-: stand-alone WHY tool, to use both automatic and interactive provers
- 2003-: KRAKATOA tool for JAVA programs
- 2004-: CADUCEUS tool for C programs
- 2007: The WHY platform

Outline

1. overview of the Why platform
2. SMT solvers and program verification
   - theories of interest for program verification
3. SMT-lib and SMT-comp

Overview of the Why Platform

- general **goal**: prove behavioral properties of **pointer programs**
- pointer program = program manipulating data structures with **in-place mutable fields**
- we currently focus on **C** and **Java** programs

What Kind of Properties

two kinds

- **safety**, that is
  - no null pointer dereference
  - no array access out of bounds (no buffer overflow)
  - no division by zero
  - no arithmetic overflow
  - termination

- **behavioral correctness**
  - the program does what it is expected to do

Principles

- specification as **annotations** at the source code level
  - Java: an extension of JML (Java Modeling Language)
  - C: our own language (mostly JML-inspired)
- generation of **verification conditions** (VCs)
  - using Hoare logic / weakest preconditions
  - similar approaches: static ESC/Java, SPEC#, B method, etc.
- **multi-prover** approach
  - off-the-shelf provers, as many as possible
  - automatic provers (Alt-Ergo, Simplify, Yices, Z3, CVC3, etc.)
  - proof assistants (Coq, PVS, Isabelle/HOL, etc.)

A Toy Example: Binary Search

binary search: search a sorted array of integers for a given value

famous example; see J. Bentley's *Programming Pearls*
most programmers are wrong on their first attempt to write binary search

Binary Search (C code)

```c
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1;
  while (l <= u ) {
    int m = (l + u) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else
      return m;
  }
  return -1;
}
```

Binary Search: Safety

- no division by zero
- no array access out of bounds
- termination

```
/*@ requires n >= 0 && \valid_range(t,0,n-1) */
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1;
  /*@ invariant 0 <= l && u <= n-1
    @ variant   u-l
    @*/
  while (l <= u ) {
    ...
}
```

DEMO

Binary Search: Behavioral Specification

```
/*@ requires
  @   n >= 0 && \valid_range(t,0,n-1) &&
  @   \forall int k1, int k2;
  @     0 <= k1 <= k2 <= n-1 => t[k1] <= t[k2]
  @ ensures
  @   (\result >= 0 && t[\result] == v) ||
  @   (\result == -1 &&
  @      \forall int k; 0 <= k < n => t[k] != v)
  @*/
int binary_search(int* t, int n, int v) {
  ...
}
```

Binary Search: Behavioral Specification (cont'd)

requires a stronger invariant

```
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1;
  /*@ invariant
    @   0 <= l && u <= n-1 &&
    @   \forall int k;
    @      0 <= k < n => t[k] == v => l <= k <= u
    @ variant u-l
    @*/
  while (l <= u ) {
    ...
  }
}
```

DEMO

Binary Search: Arithmetic Overflows

finally, let's prove that there is no arithmetic overflow... **there is one!**

in statement

```
int m = (l + u) / 2;
```

a possible overflow is signaled; a possible fix is

```
int m = l + (u - l) / 2;
```

see

- Google: "Read All About It: Nearly All Binary Searches and Mergesorts are Broken"
- "Types, Bytes, and Separation Logic" POPL'07

Binary Search: Arithmetic Overflows

finally, let's prove that there is no arithmetic overflow... **there is one!**

in statement

```
int m = (l + u) / 2;
```

a possible overflow is signaled; a possible fix is

```
int m = l + (u - l) / 2;
```

see

- Google: "Read All About It: Nearly All Binary Searches and Mergesorts are Broken"
- "Types, Bytes, and Separation Logic" POPL'07

Verification Technique

we use a standard technology (component-as-array memory model, weakest preconditions, etc.)

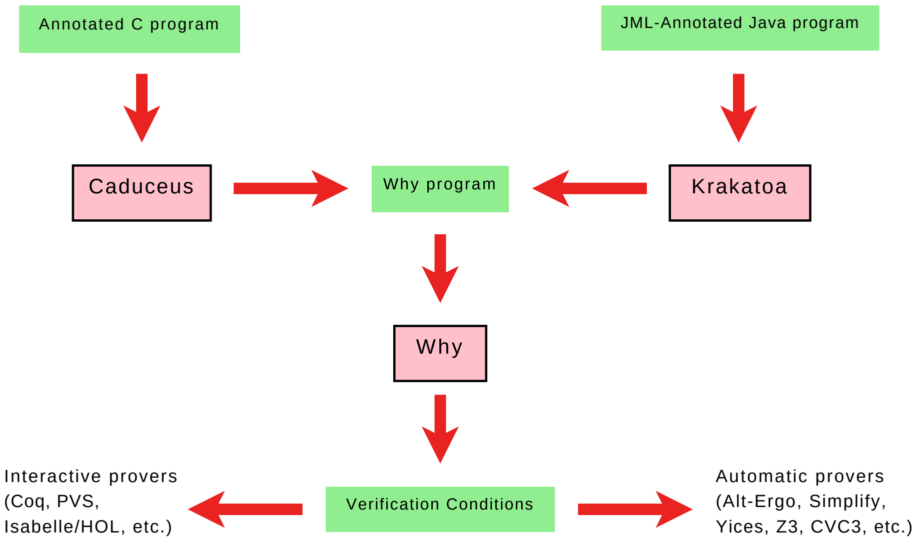
two specific issues:

- how to share the effort which is common to C and Java
- how to use many different theorem provers

our solution: the use of an intermediate language, **Why**, which is

- a VC generator
- a common front-end to various provers

Platform Overview



Why: a Verification Condition Generator

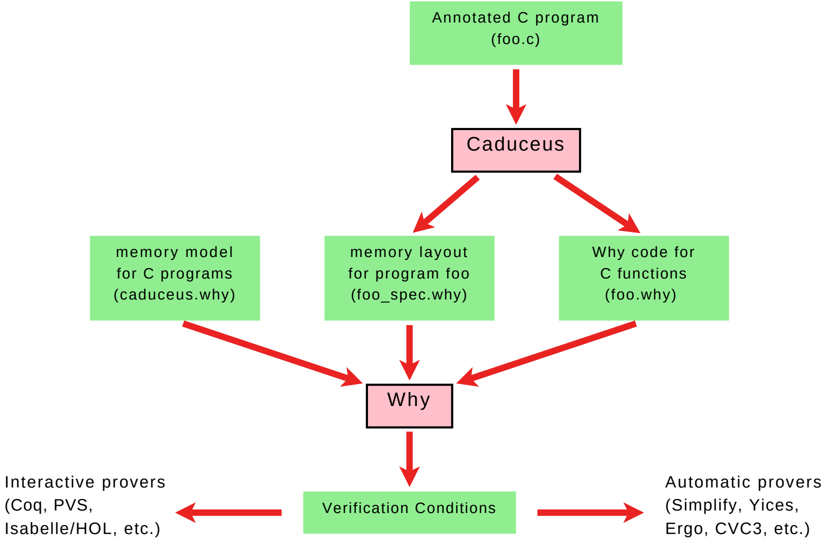
Why is a **verification condition generator** for a language with

- variables containing pure values, no alias (~ Hoare-logic language)
- usual control structures (loops, tests, etc.)
- exceptions
- (possibly recursive) functions
- polymorphic first-order logic with equality and arithmetic

Why is similar to Boogie (SPEC# project)

Why is also responsible for **translating** verification conditions to the **native logics** of all provers

Generating the Verification Conditions



SMT Solvers and Program Verification

Disclaimer

don't be mistaken by the remaining of this talk;

I do think that

- SMT solvers are great tools!
- SMT-lib is definitely a good idea
- SMT-comp helps improving the quality of SMT solvers

Which Logics for Program Verification

SMT solvers provide

- **first-order logic with equality**
  - memory models, user axiomatic models, etc.
- **integer/rational/real linear arithmetic**
  - integer arithmetic: array indices, pointer arithmetic, etc.
- **applicative arrays**
  - axiomatic approach is equally efficient
  - extensionality is not needed in practice
- **fixed-size bit vectors**
  - a too restrictive interface
- **tuples**, **records**, **inductive data types**

Which Logics for Program Verification

relevant theories for program verification can be different

- **non-linear arithmetic**
- **finite sets**
- **reachability**

let us consider some examples

- Bresenham's line drawing algorithm
- Dijkstra's shortest path algorithm

Example 1: Bresenham's Line Drawing Algorithm

draws a discrete line from $(0, 0)$ to (x_2, y_2)

```
logic x2,y2 : int
```

```
axiom first_octant : 0 <= y2 <= x2
```

`x` varies from 0 to x_2 ;
at each step, `y` is increased or not, according to the size of `e`

```
parameter x,y,e : int ref
```

Example 1: Bresenham's Line Drawing Algorithm

```
let bresenham () =
  x := 0;
  y := 0;
  e := 2 * y2 - x2;
  while !x <= x2 do
    { invariant 0 <= x <= x2 + 1 and
        e = 2 * (x + 1) * y2 - (2 * y + 1) * x2 and
        2 * (y2 - x2) <= e <= 2 * y2 }
    (* here we would plot (x,y) *)
    if !e < 0 then
      e := !e + 2 * y2
    else begin
      y := !y + 1;
      e := !e + 2 * (y2 - x2)
    end;
    x := !x + 1
  done
```

Example 1: Bresenham's Line Drawing Algorithm

the code only uses linear arithmetic

the specification and thus the proofs require **non-linear arithmetic**

if suffices to add the following axioms

```
axiom z_ring_0 : forall a,b,c: int.  a * (b+c) = a*b + a*c
axiom z_ring_1 : forall a,b,c: int.  (b+c) * a = b*a + c*a
```

DEMO

Example 2: Dijkstra's Shortest Path

single-source shortest path in a weighted graph

$S \leftarrow \emptyset$
$Q \leftarrow \{src\};\ d[src] \leftarrow 0$
while $Q \backslash S$ not empty do
  extract u from $Q \backslash S$ with minimal distance $d[u]$
  $S \leftarrow S \cup \{u\}$
  for each vertex v such that $u \xrightarrow{w} v$
    $d[v] \leftarrow \min(d[v], d[u] + w)$
    $Q \leftarrow Q \cup \{v\}$

Example 2: Dijkstra's Shortest Path

finite sets are everywhere in the code/specification:

- set of vertices V
- set of successors of u
- sets S and Q

all we need is

- the empty set $\emptyset$
- addition $\{x\} \cup s$
- subtraction $s \backslash \{x\}$
- membership predicate $x \in s$

Example 2: Dijkstra's Shortest Path

```
type 'a set

logic set_empty : 'a set
logic set_add : 'a, 'a set -> 'a set
logic set_rmv : 'a, 'a set -> 'a set
logic In : 'a, 'a set -> prop

predicate Is_empty(s : 'a set) =
  forall x: 'a.  not In(x, s)

predicate Incl(s1 : 'a set, s2 : 'a set) =
  forall x: 'a.  In(x, s1) -> In(x, s2)
```

Example 2: Dijkstra's Shortest Path

```
axiom set_empty_def :
  Is_empty(set_empty)

axiom set_add_def :
  forall x: 'a.  forall y: 'a.  forall s: 'a set.
  In(x, set_add(y,s)) <-> (x = y or In(x, s))

axiom set_rmv_def :
  forall x: 'a.  forall y: 'a.  forall s: 'a set.
  In(x, set_rmv(y,s)) <-> (x <> y and In(x, s))
```

Example 2: Dijkstra's Shortest Path

termination requires the notion of cardinality

```
logic set_card : 'a set -> int

axiom card_nonneg : forall s:'a set. set_card(s) >= 0

axiom card_set_add :
  forall x:'a. forall s:'a set.
  not In(x,s) -> set_card(set_add(x,s)) = 1 + set_card(s)

axiom card_set_rmv :
  forall x:'a. forall s:'a set.
  In(x,s) -> set_card(s) = 1 + set_card(set_rmv(x, s))

axiom card_Incl :
  forall s1,s2 : 'a set.
  Incl(s1,s2) -> set_card(s1) <= set_card(s2)
```

Example 2: Dijkstra's Shortest Path

```
while ... do
  { ... variant set_card(V) - set_card(S) }
  ...
  S := set_add u !S;
  ...
  while ... do
    { ... variant set_card(su) }
    ...
    su := set_rmv v !su
  done
done
```

Finite Sets

a theory of **finite sets** with constant $\emptyset$, operations $\{x\} \cup s$, $s \backslash \{x\}$, card(s) and predicate $x \in s$ would be extremely useful (even if incomplete)

Example 2: Dijkstra's Shortest Path

```
(* paths *)

logic path : vertex, vertex, int -> prop

axiom path_nil :
  forall x: vertex.  path(x,x,0)

axiom path_cons :
  forall x,y,z: vertex.  forall d: int.
  path(x,y,d) -> In(z,g_succ(y)) ->
  path(x,z,d+weight(y,z))

(* and shortest paths *)

predicate shortest_path(x: vertex, y: vertex, d: int) =
  path(x,y,d) and forall d': int.  path(x,y,d') -> d <= d'
```

Example 2: Dijkstra's Shortest Path

```
axiom path_inversion :
  forall src,v: vertex.  forall d: int.  path(src,v,d) ->
    (v = src and d = 0) or
    (exists v': vertex.
      path(src,v',d - weight(v',v)) and In(v,g_succ(v')))

(* lemmas requiring induction *)

axiom length_nonneg :
  forall x,y: vertex.  forall d: int.  path(x,y,d) -> d >= 0

...
```

Reachability

more generally, a theory of **reachability** is often used when specifying programs

this is simply the **reflexive transitive closure** of some relation (requires a some kind of higher-order to be generic)

variants:

- paths without repetition
- paths with the list of nodes ($path(x, y, l)$)
- closure of a functionø
- etc.

Conversely

sometimes, no need for built-in theories

examples

- arrays
- machine arithmetic (fixed-size integers)
- bitwise arithmetic (low-level bit tricks)

Machine Arithmetic

3 possible models for C (integer) arithmetic in Why

- exact arithmetic
- bounded arithmetic (no overflow)
- modulo arithmetic (faithful to program execution)

Bounded Arithmetic

```
type int32

logic of_int32: int32 -> int

axiom int32_domain :
 forall x: int32. -2147483648 <= of_int32(x) <= 2147483647

parameter int32_of_int :
  x: int ->
    { -2147483648 <= x <= 2147483647 }
    int32
    { of_int32(result) = x }
```

Bounded Arithmetic

a C operation such as $x + y$ is translated into

```
int32_of_int(of_int32(x) + of_int32(y))
```

which produces the verification condition

```
-2147483648 <= of_int32(x) + of_int32(y) <= 2147483647
```

no real need for a built-in theory

Modulo Arithmetic

```
type int32
logic of_int32: int32 -> int
axiom int32_domain : ...

logic mod_int32: int -> int

parameter int32_of_int :
  x: int -> { } int32 { of_int32(result) = mod_int32(x) }

axiom mod_int32_id :
  forall x: int.
  -2147483648 <= x <= 2147483647 -> mod_int32(x) = x

axiom mod_int32_def :
  forall x,k: int.
  mod_int32(x) = mod_int32(x + k * 4294967296)
```

Modulo Arithmetic

in some cases, you may only need

```
axiom mod_int32_gt :
  forall x: int.  x > 2147483647 ->
  mod_int32(x) = mod_int32(x - 4294967296)

axiom mod_int32_lt :
  forall x: int.  x < -2147483648 ->
  mod_int32(x) = mod_int32(x + 4294967296)
```

otherwise, you need

- either non-linear arithmetic
- or at a built-in theory of modulo arithmetic

Bitwise Arithmetic

challenge for **the verified program of the month**:

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

Unobfuscating...

```
int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d)
      f += t(a-d, (b+d)*2, (c+d)/2);
  return f;
}

int main(int q) {
  scanf("%d", &q);
  printf("%d\n", t(~(~0<<q), 0, 0));
}
```

this program reads an integer *n*
and prints the number of solutions to the *n*-queens problem

SMT-lib and SMT-comp

conclusion

Conclusion