High-Level Functional Properties of Bit-Level Programs: Formal Specifications and Automated Proofs

Claire Dross (AdaCore) Clément Fumex (Inria)
Jens Gerlach (Fraunhofer First) Claude Marché (Inria)

Gallium Seminar, April 11th, 2016

Outline

First Example

```
uint32_t f(uint32_t a) {
  return (a|-a);
}
```

- What does this code compute ?

First Example

```
uint32_t f(uint32_t a) {
  return (a|-a);
}
```

- What does this code compute ?
- Example (8 bits):

$$\begin{array}{rcl} \text{a} & = & 10101100 \\ \text{-a} & = & 01010100 \\ \hline \text{a} \mid \text{-a} & = & 11111100 \end{array}$$

First Example

```
uint32_t f(uint32_t a) {
  return (a|-a);
}
```

- What does this code compute ?
- Example (8 bits):

a	=	10101100
-a	=	01010100
a | -a	=	11111100

- Spread the *rightmost 1-bit* to the left

How to specify this code ?

```c
uint32_t f(uint32_t a) {
  return (a|-a);
}
```

Informal specification:

- An unsigned value represent a subset of $\{0..size-1\}$ with the indices of its *1-bits*
- `f a` represents the subset $\{x | x \geq min(a)\}$

Example from Esterel compiler

Challenge given by Gérard Berry.

- Instruction returns integer code between 1 and N
- Parallel execution returns maximum of codes of its branches

Example from Esterel compiler

Challenge given by Gérard Berry.

- Instruction returns integer code between 1 and N
- Parallel execution returns maximum of codes of its branches
- Static analysis: each instruction P may return a set of codes $C(P)$ instead of one code only
- Hence $P||Q$ return

$$\{max(p, q) | p \in C(P), q \in C(Q)\}$$

Example from Esterel compiler

Challenge given by Gérard Berry.

- Instruction returns integer code between 1 and N
- Parallel execution returns maximum of codes of its branches
- Static analysis: each instruction P may return a set of codes $C(P)$ instead of one code only
- Hence $P||Q$ return

$$\{max(p,q)|p \in C(P), q \in C(Q)\}$$

- Return codes are implemented as *bit-vectors*
- $C(P||Q)$ can be computed as

$$(P|Q)\&(P|-P)\&(Q|-Q)$$

[Gonthier]

Motivations

- We want to *formally specify* codes that *mix bitwise operators and integer arithmetic*
- The *specifications* should be at an *abstract mathematical level*
- We want the *proofs* to be as *automatic* as possible.

Summary of our approach

Why3

- Environment for Deductive Verification
- `http://why3.lri.fr`

- We designed a *rich theory of bitvectors*
  - used both for specification and code
- We use the *built-in bitvector theory* provided by some *SMT solvers* (CVC4, Z3)

SPARK 2014

- Development of safety-critical Ada programs
- Proof of contracts via Why3

- *Modular types* encoded as Why3's bitvector types

Outline

First example: formal specification in Why3

```
use import bv.BV32
use import int.Int
use import set.Fsetint
(* a 32-bit bitvector and its interpretation as a set *)
type s = { bv : t; ghost mdl: set int; }
invariant
 { forall i: int. (0 ≤ i < size ∧ nth self.bv i) ↔ mem i self.mdl }

let aboveMin (a : s) : s (* operator [a|-a] *)
  requires { not is_empty a.mdl }
  ensures  { result.mdl = interval (min_elt a.mdl) size }
= ...
```

Module BV32 provides in particular:

- type t: bitvectors of size 32
- nth x n : the n-th bit of x is a 1

First example: formal specification in Why3

```
use import bv.BV32
use import int.Int
use import set.Fsetint
(* a 32-bit bitvector and its interpretation as a set *)
type s = { bv : t; ghost mdl: set int; }
invariant
 { forall i: int. (0 ≤ i < size ∧ nth self.bv i) ↔ mem i self.mdl }

let aboveMin (a : s) : s (* operator [a|-a] *)
  requires { not is_empty a.mdl }
  ensures  { result.mdl = interval (min_elt a.mdl) size }
= ...
```

Remark

Naturally, specifications will mix:

- bitvectors
- mathematical integers
- and any other theories, such as Fsetint

First example: code

```
let aboveMin (a : s) : s (* operator [a|-a] *)
  requires { not is_empty a.mdl }
  ensures  { result.mdl = interval (min_elt a.mdl) size }
= let ghost p = min_elt a.mdl in
  let res = bw_or a.bv (neg a.bv) in
  { bv = res;
    mdl = interval p size }
```

More operators

- `bw_or x y`: bitwise or
- `neg x`: arithmetic negation (2-complement)

First example: the proof

```
let aboveMin (a : s) : s
  requires { not is_empty a.mdl }
  ensures  { result.mdl = interval (min_elt a.mdl) size }
= let ghost p = min_elt a.mdl in
  assert { eq_sub a.bv zeros 0 p };
  let res = bw_or a.bv (neg a.bv) in
  assert { eq_sub res zeros 0 p };
  assert { eq_sub res ones p (size - p) };
  { bv = res;
    mdl = interval p size }
```

a	=	10101100
-a	=	01010100
a | -a	=	11111100

Additional predicate and operators

- `of_int i`: `i` interpreted as a bit-vector
- `eq_sub a b i l`: all the bits of a and b between positions i and $i+l-1$ are equal
- `zeros`, `ones`: the bitvectors with all bits set to 0, resp. 1

First example: the proof

```
let aboveMin (a : s) : s
  requires { not is_empty a.mdl }
  ensures  { result.mdl = interval (min_elt a.mdl) size }
= let ghost p = min_elt a.mdl in
  assert { eq_sub a.bv zeros 0 p };
  let res = bw_or a.bv (neg a.bv) in
  assert { eq_sub res zeros 0 p };
  assert { eq_sub res ones p (size - p) };
  { bv = res;
    mdl = interval p size }
```

min_elt axiomatization

```
axiom min_elt_def2:
  forall s: set int. forall x: int. mem x s → min_elt s ≤ x
```

First example: the proof

```
let aboveMin (a : s) : s
  requires { not is_empty a.mdl }
  ensures  { result.mdl = interval (min_elt a.mdl) size }
= let ghost p = min_elt a.mdl in
  let ghost p_bv = of_int p in
  assert { eq_sub_bv a.bv zeros zeros p_bv };
  let res = bw_or a.bv (neg a.bv) in
  assert { eq_sub_bv res zeros zeros p_bv };
  assert { eq_sub_bv res ones p_bv (sub size_bv p_bv) };
  { bv = res;
    mdl = interval p size }
```

“bv” variants of operator

```
eq_sub_bv a b i l = eq_sub a b (to_uint i) (to_uint l)
...
```

First example: Proof Results

Proof obligations	Alt-Ergo (1.01)	CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.2)	Z3 (4.4.2 noBV)
1. assertion	0.20	3.01	0.07	(5s)	(5s)
2. assertion	(5s)	0.37	2.48	0.49	(5s)
3. assertion	(5s)	0.82	2.72	(5s)	(5s)
4. type invariant	0.62	3.90	0.27	(5s)	(5s)
5. postcondition	0.03	0.04	0.04	0.01	0.01

Need for proofs

Two different drivers for the same prover

Berry's challenge: formal specification in Why3

```
use import int.MinMax

let maxUnion (a b : s) : s  (* operator [(a|b)&(a|-a)&(b|-b)] *)
  requires { not is_empty a.mdl ∧ not is_empty b.mdl }
  ensures { forall x. mem x result.mdl ↔
              exists y z. mem y a.mdl ∧ mem z b.mdl ∧ x = max y z }
= ...
```

Berry's challenge: code

```
let maxUnion (a b : s) : s  (* operator [(a|b)&(a|-a)&(b|-b)] *)
  requires { not is_empty a.mdl ∧ not is_empty b.mdl }
  ensures  { forall x. mem x result.mdl ↔
                exists y z. mem y a.mdl ∧ mem z b.mdl ∧ x = max y z }
=
   intersection (union a b) (intersection (aboveMin a) (aboveMin b))
```

Remark

No bit manipulation at this stage

Berry's challenge: code

```
let maxUnion (a b : s) : s  (* operator [(a|b)&(a|-a)&(b|-b)] *)
  requires { not is_empty a.mdl ∧ not is_empty b.mdl }
  ensures  { forall x. mem x result.mdl ↔
               exists y z. mem y a.mdl ∧ mem z b.mdl ∧ x = max y z }
=
   intersection (union a b) (intersection (aboveMin a) (aboveMin b))
```

```
let union (a b: s) : s  (* operator [a|b] *)
  ensures  { result.mdl = union b.mdl a.mdl }
= { bv = bw_or a.bv b.bv;
    mdl = union b.mdl a.mdl }

let intersection (a b : s) : s (* operator [a&b] *)
  ensures { result.mdl = inter a.mdl b.mdl }
= { bv = bw_and a.bv b.bv;
    mdl = inter a.mdl b.mdl }
```

Berry's challenge: proof

```
let maxUnion (a b : s) : s  (* operator [(a|b)&(a|-a)&(b|-b)] *)
  requires { not is_empty a.mdl ∧ not is_empty b.mdl }
  ensures { forall x. mem x result.mdl ↔
              exists y z. mem y a.mdl ∧ mem z b.mdl ∧ x = max y z }
= let res =
    intersection (union a b) (intersection (aboveMin a) (aboveMin b))
  in
  assert {
    forall x. mem x res.mdl →
      let (y,z) =
        if mem x a.mdl then (x,min_elt b.mdl) else (min_elt a.mdl,x)
      in
        mem y a.mdl ∧ mem z b.mdl ∧ x = max y z };
  res
```

Berry's challenge: Proof Results

Proof obligations	Alt-Ergo (1.01)	CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.2)	Z3 (4.4.2 noBV)
VC for union	0.24	3.87	0.06	(5s)	(5s)
VC for intersection	0.24	3.73	0.06	(5s)	(5s)
1. precondition	0.02	0.06	0.04	0.01	0.01
2. precondition	0.02	0.07	0.03	0.01	0.01
3. assertion	0.34	0.25	0.20	0.81	(5s)
4. postcondition	(5s)	0.06	0.12	(5s)	(5s)
	0.26	0.15	0.11	(5s)	(5s)

Summary of identified needs

Need for *two abstraction levels* in the Why3 theory of bit-vectors

At the level of bits:

- operators `eq_sub_bv` and other "bv" variants
  - for low-level assertions
- variant "BV" of drivers
  - to exploit native support of SMT solvers

At a more abstract level

- operators `nth`, `eq_sub`, ...
- variant "noBV" of drivers

Outline

SMTLIB theory for bitvectors

The SMTLIB standard propose a theory of *fixed-size bitvectors*

- A *family of sorts* `(_ BitVec` i`)` for any numeral $i > 0$
- Binary operators on bitvectors of same size
  `bvand`, `bvor`, `bvadd`, `bvmul`, `bvudiv`, `bvsrem`, `bvshl`, `bvlshr`, etc.
- `concat` : concatenates two bitvectors
- `(_ extract` i j`)`: extract sub-bitvector from bit j to i

Not abstract enough

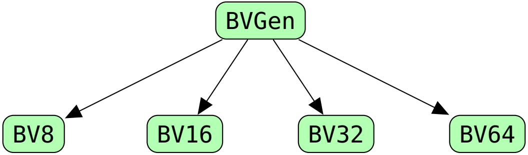
Too close to the processor's Arithmetic Logic Unit

Our Theory

- A *generic theory* with a parameter `size`.

```
constant size : int          (* size of bitvectors *)

axiom size_pos : size > 0

type t                        (* abstract type of bitvectors *)
```

- *Cloned* for sizes 8, 16, 32 and 64



Axiomatization strategy

The base operator nth is declared abstract

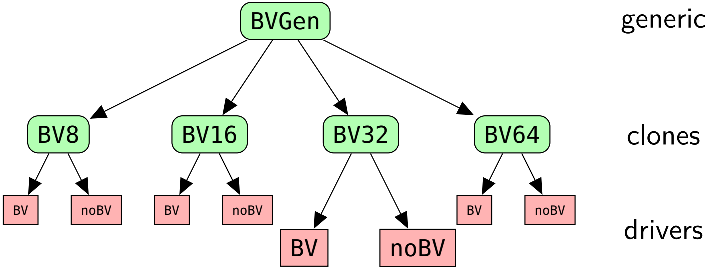
```
function nth t int : bool
```

and used to axiomatize the bitwise operators

Example

```
function bw_and t t : t

axiom bw_and_spec:
  forall v1 v2:t, n:int. 0 ≤ n < size →
    nth (bw_and v1 v2) n = andb (nth v1 n) (nth v2 n)
```

Mapping to provers



	BV	noBV
type t	(_ BitVec 32)	abstract
nth	uninterpreted	uninterpreted
bw_and	bvand	uninterpreted
bw_and_spec	removed	kept

Shift Operators

```
function lsr t int : t

axiom lsr_spec_low:
  forall b:t,n s:int. 0 ≤ s → 0 ≤ n → n+s < size →
    nth (lsr b s) n = nth b (n+s)
axiom lsr_spec_high:
  forall b:t,n s:int. 0 ≤ s → 0 ≤ n → n+s ≥ size →
    nth (lsr b s) n = False
```

	BV	noBV
lsr	uninterpreted	uninterpreted
lsr_spec_low	removed	kept
lsr_spec_high	removed	kept

Conversion with Integers

`to_uint` is supposed to map $b_{n-1} \cdots b_1 b_0$ to $\sum_{i=0}^{n-1} b_i \times 2^i$

Not fully axiomatized: we do not want provers to reason about this computation

```
constant two_power_size : int = pow2 size

function to_uint t : int

axiom to_uint_extensionality :
  forall v,v':t. to_uint v = to_uint v' → v = v'

axiom to_uint_bounds :
  forall v:t. 0 ≤ to_uint v < two_power_size
```

Arithmetic Operators

Arithmetic operators are axiomatized through `to_uint`.

```
function add t t : t

axiom add_spec: forall x y:t.
  to_uint (add x y) = mod (to_uint x + to_uint y) two_power_size

lemma add_bounded: forall v1 v2.
  to_uint v1 + to_uint v2 < two_power_size →
  to_uint (add v1 v2) = to_uint v1 + to_uint v2
```

	BV	noBV
add	bvadd	uninterpreted
add_spec	removed	removed
add_bounded	removed	kept

BV version of some operators

```
function nth_bv t t : bool

axiom nth_bv_def:
  forall x i. nth_bv x i = not (bw_and (lsr_bv x i) (of_int 1) = zeros)

axiom nth_bv_is_nth:
  forall x i: t. nth_bv x i = nth x (to_uint i)

function lsr_bv t t : t

axiom lsr_bv_is_lsr: forall x n. lsr_bv x n = lsr x (to_uint n)
```

	BV	noBV
nth_bv	uninterpreted	uninterpreted
nth_bv_def	kept	removed
nth_bv_is_nth	kept	kept
lsr_bv	bvlshr	uninterpreted
lsr_bv_is_lsr	kept	kept

Sub Equality

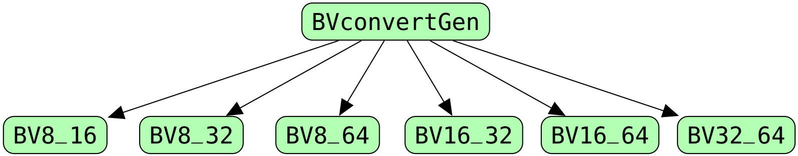
```
predicate eq_sub (a b:t) (i n:int) =
  forall j:int. i ≤ j < i + n → nth a j = nth b j

predicate eq_sub_bv t t t t

axiom eq_sub_bv_def:
  forall a b i n:t.
    let mask = lsl_bv (sub (lsl_bv (of_int 1) n) (of_int 1)) i
    in
      eq_sub_bv a b i n = (bw_and b mask = bw_and a mask)

lemma eq_sub_equiv: forall a b i n:t.
  eq_sub a b (to_uint i) (to_uint n) ↔ eq_sub_bv a b i n
```

	BV	noBV
eq_sub	uninterpreted	uninterpreted
eq_sub_bv	uninterpreted	uninterpreted
eq_sub_bv_def	kept	removed
eq_sub_equiv	kept	kept

Conversions

Conversions

A generic theory cloned for the 6 possible conversion configurations.



BV32_64	BV	noBV
toBig	(_ zero_extend 32)	uninterpreted
toSmall	(_ extract 31 0)	uninterpreted
axioms	removed	kept

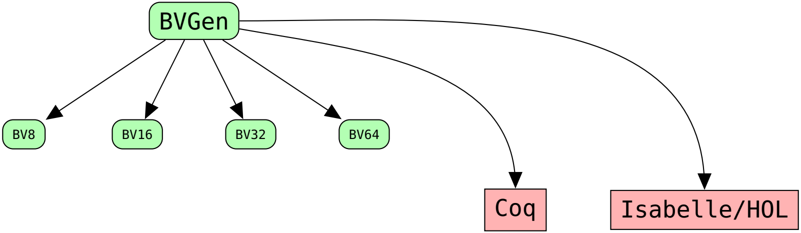
About Soundness of the theory

Coq realization

The realization of the theory is based on boolean vectors in Coq.

Isabelle realization

The realization of the theory is based on words in Isabelle/HOL



Outline

What is SPARK?

SPARK is a programming language

- Subset of Ada, a language targeted at reliable embedded software
- Designed for formal analysis (flow analysis, proof)

What is SPARK?

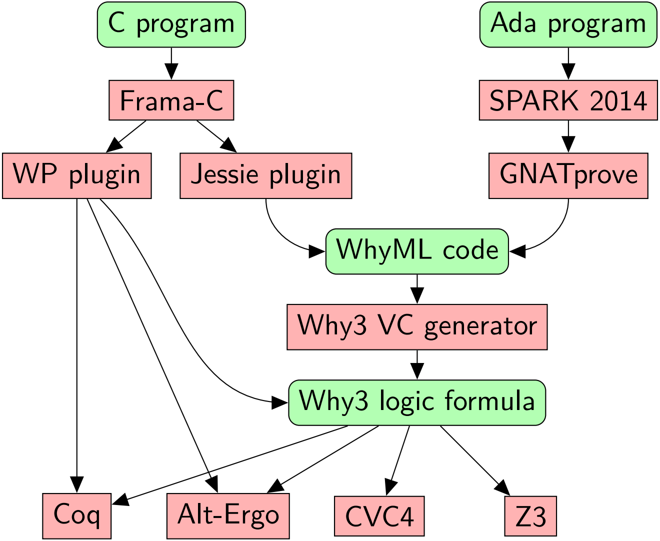
SPARK is a programming language

- Subset of Ada, a language targeted at reliable embedded software
- Designed for formal analysis (flow analysis, proof)

Proof of SPARK programs:

- GNATprove tool
- Uses WhyML as an intermediate language

What is GNATprove?



Modular Types in Ada

```
type M is mod I;
```

- type of unsigned integers in `{0 .. I - 1}`
- modular semantic
- *no overflow, no runtime error* (besides division by zero)

Modular Types in Ada

```
type M is mod I;
```

- type of unsigned integers in `{0 .. I - 1}`
- modular semantic
- *no overflow, no runtime error* (besides division by zero)

```
type M is mod I range A .. B;
```

- run time error if outside range at assignment and parameter passing

Translation to Why3

```
type BV8 is mod 2**8;
```

mapped to the corresponding bitvector type `BV8.t`

Translation to Why3

```
type BV8 is mod 2**8;
```

mapped to the corresponding bitvector type `BV8.t`

```
A : BV8 := 42;
B : BV8 range 1 .. 10 := A + 1;
```

translated to

```
let a : t = of_int 42 in
let b : t = add a (of_int 1) in
assert { ule (of_int 1) b ∧ ule b (of_int 10) };
...
```

Translation to Why3

```
type BV8 is mod 2**8;
```

mapped to the corresponding bitvector type `BV8.t`

```
A : BV8 := 42;
B : BV8 range 1 .. 10 := A + 1;
```

translated to

```
let a : t = of_int 42 in
let b : t = add a (of_int 1) in
assert { ule (of_int 1) b ∧ ule b (of_int 10) };
...
```

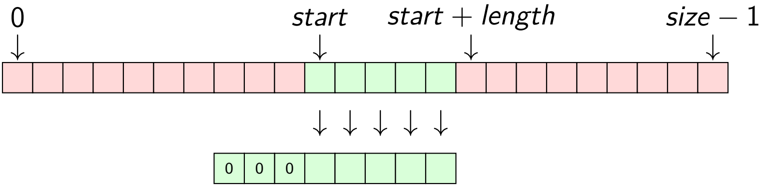
Extends to builtins: shifts/rotates, logical operations...

Outline

Case Study: BitWalker

- Original C version provided by Siemens in the context of the ITEA 2 project OpenETCS [http://openetcs.org]
- Rewritten by Jens Gerlach for Frama-C/WP
- *Formal specification in ACSL*
- Formal specification relies on a *Coq bitvector theory*
- A significant part of the proofs are done *interactively within Coq*

BitWalker functionality



- Peek: Read data from a stream of bits to a 64bit integer
- Poke: Converse operation

Specifics

- stream encoded as array of bytes
- bit 0 at left
- only unsigned integers and bitwise / shift operations

Peek C Code

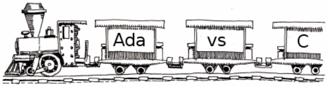
```
uint64_t Bitwalker_Peek(uint32_t  start, uint32_t  length,
                        uint8_t*  addr, uint32_t  size) {
  if ((start + length) > 8 * size) return 0;
  uint64_t retval = 0;
  for (uint32_t i = 0; i < length; i++) {
    int flag = PeekBit8Array(addr, size, start + i);
    retval = PokeBit64(retval, 64u - length + i, flag);
  }
  return retval;
}
```

Peek C Code

```
uint64_t Bitwalker_Peek(uint32_t  start, uint32_t  length,
                        uint8_t*  addr, uint32_t  size) {
  if ((start + length) > 8 * size) return 0;
  uint64_t retval = 0;
  for (uint32_t i = 0; i < length; i++) {
    int flag = PeekBit8Array(addr, size, start + i);
    retval = PokeBit64(retval, 64u - length + i, flag);
  }
  return retval;
}
```

```
//sets the bit at index [left] in [value] to the value of [flag]
uint64_t PokeBit64(uint64_t value, uint32_t left, int flag) {
  uint64_t mask = ((uint64_t) 1u) << (63 - left);
  return (flag == 0) ? (value & ~mask) : (value | mask);
}
```

C to Ada Interlude



Ada code is “more strongly” typed

- Ada has Booleans!
- No implicit conversion between integer and modular type
  - except with explicit sub-typing, e.g.

    ```
    subtype Natural is Integer range 0 .. Integer'Last;
    ```

  - Array indexes are typically naturals, not modular
- Shift operator in Ada:
  - first argument: modular (i.e. “unsigned”)
  - second argument: integer (i.e. “signed”).

PokeBit64 code: C vs Ada

```c
//sets the bit at index [left] in [value] to the value of [flag]
uint64_t PokeBit64(uint64_t value, uint32_t left, int flag) {
  uint64_t mask = ((uint64_t) 1u) << (63 - left);
  return (flag == 0) ? (value & ~mask) : (value | mask);
}
```

```ada
function PokeBit64 (Value : Unsigned_64; Left : Natural;
                    Flag : Boolean) return Unsigned_64 is
   Mask : constant Unsigned_64 := Shift_Left (1, 63 - Left);
begin
   return (if Flag then (Value or Mask)
                   else (Value and (not Mask)));
end PokeBit64;
```

PokeBit64 specification: C vs Ada

```
/*@ requires left < 64;
  @ ensures \forall integer i; (0 ≤ i < 64  &&  i != left) ==>
  @            (LeftBit64(\result, i) ≤=> LeftBit64(value, i));
  @  ensures flag != 0  ≤=>  LeftBit64(\result, left);
  @*/
uint64_t PokeBit64(uint64_t value, uint32_t left, int flag);
```

```
function PokeBit64 (Value : Unsigned_64; Left  : Natural;
                    Flag  : Boolean) return Unsigned_64
with
  Pre => Left < 64,
  Post => (for all I in Natural range 0 .. 63 =>
           (if I /= 63 - Left then
            Nth (PokeBit64'Result, I) = Nth (Value, I)))
    and (Flag = Nth (PokeBit64'Result, 63 - Left));
```

Proving PokeBit64

```
function PokeBit64 (Value : Unsigned_64; Left  : Natural;
                    Flag  : Boolean) return Unsigned_64
with
  Pre => Left < 64,
  Post => (for all I in Natural range 0 .. 63 =>
            (if I /= 63 - Left then
             Nth (PokeBit64'Result, I) = Nth (Value, I)))
    and (Flag = Nth (PokeBit64'Result, 63 - Left));
```

```
function PokeBit64 (Value : Unsigned_64; Left : Natural;
                    Flag : Boolean) return Unsigned_64 is
   Mask : constant Unsigned_64 := Shift_Left (1, 63 - Left);
   R    : constant Unsigned_64 := (if Flag then (Value or Mask)
                                   else (Value and (not Mask)));
begin
   return R;
end PokeBit64;
```

Proving PokeBit64

```
function PokeBit64 (Value : Unsigned_64; Left : Natural;
                    Flag : Boolean) return Unsigned_64 is
   Mask : constant Unsigned_64 := Shift_Left (1, 63 - Left);
   R    : constant Unsigned_64 := (if Flag then (Value or Mask)
                                   else (Value and (not Mask)));
begin
   pragma Assert (Left < 64);
   pragma Assert (for all I in Natural range 0 .. 63 =>
         (if I /= 63 - Left then
          Nth (R, I) = Nth (Value, I)));

   pragma Assert (Flag = Nth (R, 63 - Left));
   return R;
end PokeBit64;
```

Proving PokeBit64

```
function PokeBit64 (Value : Unsigned_64; Left : Natural;
                    Flag : Boolean) return Unsigned_64 is
   Left_Bv : constant Unsigned_64 := Unsigned_64(Left) with Ghost;
   Mask : constant Unsigned_64 := Shift_Left (1, 63 - Left);
   R    : constant Unsigned_64 := (if Flag then (Value or Mask)
                                   else (Value and (not Mask)));
begin
   pragma Assert (Left_Bv < 64);
   pragma Assert (for all I in Unsigned_64 range 0 .. 63 =>
         (if I /= 63 - Left_Bv then
          Nth_Bv (R, I) = Nth_Bv (Value, I)));

   pragma Assert (Flag = Nth_Bv (R, 63 - Left_Bv));
   return R;
end PokeBit64;
```

Proving PokeBit64

```
function PokeBit64 (Value : Unsigned_64; Left : Natural;
                    Flag : Boolean) return Unsigned_64 is
   Left_Bv : constant Unsigned_64 := Unsigned_64(Left) with Ghost;
begin
   pragma Assert (Left_Bv < 64);
   pragma Assert (63 - Left_Bv = Unsigned_64 (63 - Left));
   declare
      Mask : constant Unsigned_64 := Shift_Left (1, 63 - Left);
      R : constant Unsigned_64 := (if Flag then (Value or Mask)
                                        else (Value and (not Mask)));
   begin
      pragma Assert (for all I in Unsigned_64 range 0 .. 63 =>
                       (if I /= 63 - Left_Bv
                        then Nth_Bv (R, I) = Nth_Bv (Value, I)));
      pragma Assert (for all I in Natural range 0 .. 63 =>
                       (0 <= Unsigned_64 (I) and then
                        Unsigned_64 (I) <= 63));
      pragma Assert (Flag = Nth_Bv (R, 63 - Left_Bv));
      return R;
   end;
end PokeBit64;
```

Proof results

Proof obligations	CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.0)	Z3 (4.4.0 noBV)	altergo (0.99.1)
1. assertion	6.52	0.06	(5s)	0.78	0.06
2. range check	0.07	0.06	0.01	0.18	0.03
3. assertion	(5s)	0.07	(5s)	(5s)	0.53
4. range check	0.12	0.06	0.02	0.19	0.06
5. assertion	(5s)	4.30	0.29	(5s)	(5s)
6. assertion	(5s)	0.09	(5s)	(5s)	1.22
7. assertion	0.77	(5s)	0.07	(5s)	(5s)
8. range check	0.02	0.03	0.01	0.01	0.06
9. postcondition	(5s)	0.08	(5s)	(5s)	(5s)

A peek at Peek's specification

```
function Peek  (Start, Length : Natural; Addr : Byte_Sequence)
  return Unsigned_64
with
  Pre => Addr'First = 0 and then
         Length ≤ 64 and then
         Start + Length ≤ Natural'Last and then
         8 * Addr'Length ≤ Natural'Last,
  Contract_Cases =>  (
         Start + Length > 8 * Addr'Length => Peek'Result = 0,
         Start + Length ≤ 8 * Addr'Length =>
           (for all I in 0 .. Length - 1 =>
              Nth8_Stream (Addr, Start + Length - I - 1)
            = Nth (Peek'Result, I))
          and then
           (for all I in Length .. 63 => not Nth (Peek'Result, I)));
```

- The code calls `PokeBit64` and other auxiliary functions
- The proofs don't need to speak of the bitvector level
  - No need for driver variants "BV" of CVC4 and Z3

A peek at Peek's body

```
function Peek (Start, Length : Natural; Addr : Byte_Sequence)
  return Unsigned_64 is
begin
  if Start + Length > 8 * Addr'Length then
    return 0;
  end if;
  declare
    Retval : Unsigned_64 := 0;
    Flag   : Boolean;
  begin
    for I in 0 .. Length - 1 loop
      pragma Loop_Invariant
        (for all J in Length - I .. Length - 1 =>
           Nth8_Stream (Addr, Start + Length - J - 1) = Nth (Retval, J));
      pragma Loop_Invariant
        (for all J in Length .. 63 => not Nth (Retval, J));
      Flag := PeekBit8Array (Addr, Start + I);
      Retval := PokeBit64 (Retval, (64 - Length) + I, Flag);
    end loop;
    return Retval;
  end;
end Peek;
```

Outline

Conclusions

- Automatic proofs
  - but do need hints as assertions
  - ghost code in most complex cases (as usual...)
- Abstraction:
  - Complex proofs about bitvectors done on low level functions
  - High level function specs and proofs don't speak about bitvectors
- In SPARK2014:
  - Distributed since middle 2015
  - First users seem happy:
    - VCs that were not proved before are now proved
    - Typically these are only range checks

Future Work

- In progress: similar approach for floating-point numbers
  - trying to exploit new support for FP in SMT-LIB
- Reconsider translating (signed) Integers to bitvectors ?
- How to avoid having two variants of drivers ?
  - Need of "BV" variants of drivers only for low-level code
  - It is an abstraction issue