HAL
open science

Specification and Proof of High-Level Functional Properties of Bit-Level Programs

Clément Fumex, Claire Dross, Jens Gerlach, Claude Marché

To cite this version:

Clément Fumex, Claire Dross, Jens Gerlach, Claude Marché. Specification and Proof of High-Level Functional Properties of Bit-Level Programs. NASA Formal methods, Jun 2016, Minneapolis, United States. hal-01314876

HAL Id: hal-01314876

https://inria.hal.science/hal-01314876v1

Submitted on 12 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Specification and Proof of High-Level Functional Properties of Bit-Level Programs ⋆

Clément Fumex[1,2,3], Claire Dross[3], Jens Gerlach[4], and Claude Marché[1,2]

[1] Inria, Université Paris-Saclay, F-91893 Palaiseau
[2] LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay
[3] AdaCore, F-75009 Paris
[4] Fraunhofer FOKUS, Berlin, Germany

Abstract. In a computer program, basic functionalities may be implemented using bit-wise operations. To formally specify the expected behavior of such a low-level program, it is desirable that the specification should be at a more abstract level. Formally proving that low-level code conforms to a higher-level specification is challenging, because of the gap between the different levels of abstraction. We address this challenge by designing a rich formal theory of fixed-sized bit vectors, which on the one hand allows a user to write abstract specifications close to the human—or mathematical—level of thinking, while on the other hand permits a close connection to decision procedures and tools for bit vectors, as they exist in the context of the Satisfiability Modulo Theory framework. This approach is implemented in the Why3 environment for deductive program verification, and also in its front-end environment SPARK for the development of safety-critical Ada programs. We report on several case studies used to validate our approach.

1 Introduction

It is quite common in computer programs that some basic functionality is implemented, for efficiency reasons, using bit-wise operations. There is even a famous book, *Hacker's delight* [24], which is dedicated only to this kind of smart and efficient code.

An extreme example is the following 2-line C program (a so-called "signature program" designed by Marcel van Kervinc, http://www.iwriteiam.nl/SigProgC.html).

```
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));}
```

It reads an integer n and prints another integer $f(n)$. Assuming n is smaller than the machine word size in bits (say 32), then $f(n)$ appears to be the number of solutions to the *n-queens problem*: the number of ways of placing n queens on a $n \times n$ chessboard so that they do not threaten each other. Even more remarkable, this program implements the most efficient algorithm known so far to solve this problem.

Solving the n-queens problem was used in the past as a challenge for *deductive program verification*. The challenge is to attach to such code a formal specification,

⋆ Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French national research organization

expressing its expected behavior at an abstract mathematical level (*i.e.* expressing that it really computes the number of solutions to the n-queens problem), and to prove formally that the code respects this specification. The solutions presented by Filliâtre [15], and other authors for a simplified version computing only the first solution [18], considered more abstract implementations, that do not operate directly on bits.

Deductive program verification typically proceeds by generating, from both the code and the formal specification, a set of logic formulas. These are called *verification conditions* because if one proves they are all tautologies, then the program is guaranteed to respect its specification. In program verification environments like Dafny [19] and Why3 [7], verification conditions are discharged using theorem provers, in particular those of the *Satisfiability Modulo Theories* (SMT) family such as Alt-Ergo [6], CVC4 [3], and Z3 [22]. The SMT approach is very promising for one who seeks to verify programs operating at the level of bits, because, in this context, theories for fixed-size bit vectors have been investigated for quite a long time and efficient decision procedures are known [12,4,10]. The SMT-LIB international initiative (`http://smtlib.cs.uiowa.edu/`) aims at providing standard languages and descriptions of theories for interacting with SMT solvers. SMT-LIB provides a fairly rich standard theory for fixed-size bit vectors, and decision procedures for this theory are implemented in several SMT solvers, including CVC4 and Z3.

Our objective is to add support for bit-wise operations in Why3 and its front-end SPARK2014 [21] that deals with safety-critical Ada programs. In particular, we want to exploit the bit vector decision procedures provided by SMT solvers. However, in such a context, bit-wise operations are mixed with other objects occurring in programs and specifications, such as unbounded integers, arrays, and records. We need to rely on other theories supported by SMT solvers, and also on their support for quantified axioms. Exploiting an SMT solver when several theories are mixed together with quantified axioms requires special care. This paper reports on our design choices and on some experiments we made. We start in Section 2 by illustrating our approach on a short (although non-trivial) example. In Section 3 we describe the theories for bit vectors we designed for use in Why3. In Section 4 we present how our Why3 theories are exploited in the SPARK2014 front-end. In Section 5 we illustrate our approach on a case study originating from industrial code. Our developments are distributed in SPARK Pro 16.0 and will be in the release 0.87 of Why3. More details and more case studies (including the 2-line n-queens program) are discussed in a technical report [14] and the files for the case studies are available on Toccata's Web gallery of verified programs (`http://toccata.lri.fr/gallery/bitwise.en.html`).

2 Illustrative Example

We want to specify, at an abstract level, programs that directly manipulate bits. Our approach is to exploit in parallel the theory of bit vectors supported by SMT-solvers, and their support for arithmetic and quantifiers. We provide a theory that allows the use of both on the same program. In order to do so, the intended methodology to use this theory is to specify programs at an abstract level, closer to the human mind, *e.g.* with mathematical integers, while at the same time exploiting the bit vector theories of

SMT solvers, by providing explicit hints for provers (typically under the form of extra assertions in the code) when it is necessary to help them to make the appropriate bridge between the bit vector level and the abstract level.

Let us consider an example from the Esterel compiler [5]. Each instruction returns an integer code between 1 and a fixed N. Parallel execution returns the maximum of the codes of its branches. A static analysis approximates programs by considering the set $\overline{P}$ of all possible return codes of P. Hence $\overline{P||Q} = \{max(p,q) | p \in \overline{P}, q \in \overline{Q}\}$. Sets of return codes are implemented as bit vectors, a 1 at position i in $\overline{P}$ meaning that $i \in \overline{P}$. It was suggested by Gonthier that $\overline{P||Q}$ can be computed as $(\overline{P}|\overline{Q})\&(\overline{P}|-\overline{P})\&(\overline{Q}|-\overline{Q})$.

We want to formally specify this behavior at an abstract level, not using any low-level operation like a bit-wise 'and'. Let us consider the case where $N = 32$.

Formal specification. Figure 1 presents how this code is formally specified in our setting (see [14] for details on Why3's syntax). The `use` declarations import the theory of 32-bit bit vectors we designed and the theory of finite set of integers from the Why3 library. From the former theory we use the type `t` of bit vectors, and the operator `nth`: `nth x n` is the `n`-th bit of `x` as a Boolean.

We want to relate a bit vector to its abstract view as a set of integers. We introduce a record type `s` with a field `bv : t`, and a ghost field `mdl : set int` a set of integers. A *type invariant* specifies that for each `a : s` the elements of `a.mdl` are the indexes of the 1-bits in `a.bv`. The precondition requires of `maxUnion` that the inputs are not zeros. The postcondition formalizes the former informal specification. The important point is that the formal specification is at an abstract mathematical level which is quite far from the code in the body of the function. Proving that the code satisfies the specification is thus a difficult task.

```
use import bv.BV32
use import set.FSetInt

type s = { bv : t; ghost mdl: set int; }
invariant { forall i: int.
    (0 ≤ i < size ∧ nth self.bv i)
 ↔ mem i self.mdl }

let maxUnion (a b : s) : s
  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 }
= ...
```

Fig. 1. maxUnion: formal specification

Proof. The code of `maxUnion` is split in three sub-functions shown in Figure 2. It makes use of additional operations:

- `of_int x`: integer `x` converted to a bit vector
- `eq_sub_bv x y i l`: means that the bits of a and b between positions i and $i+l-1$ are equal
- `bw_or, bw_and, neg, sub`: bit-wise and arithmetic operators on bit vectors
- `min_elt a`: the minimal element of `a`
- `interval i j`: the set $\{i \ldots j-1\}$

We emphasize that the code of `aboveMin` contains three assertions involving only bit vectors and bit-wise operators. This form of intermediate assertion is an example of a general strategy that we explain in Section 3.3.

```
let aboveMin (a : s) : s
  requires { not is_empty a.mdl }
  ensures { result.mdl = interval (min_elt a.mdl) 32 }
= 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                        (* a | -a *)
  assert { eq_sub_bv res zeros zeros p_bv };
  assert { eq_sub_bv res ones p_bv (sub (of_int 32) p_bv) };
  { bv = res; mdl = interval p 32 }

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 }

let maxUnion (a b : s) : s
  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
```

Fig. 2. maxUnion: annotated code

The proof results are displayed in Figure 3. A red background indicates an unsuccessful proof, (10m) meaning that the timeout of 10 minutes is reached, (6G) meaning that the memory limit of 6GB is reached. We stress that we use CVC4 and Z3 in two different modes. The default mode exploits their native support for bit vectors, whereas the other mode, nicknamed 'noBV' for 'no bit vectors', does not. The two VCs, 2 and 3 for `aboveMin`, are proved using the native bit vector support. On the contrary VCs 1 and 4 for `aboveMin` and the VCs for `union` and `intersection` are proved only in the mode not using native support. This need for two modes for one prover shows up in all the case studies that we considered [14]. We detail the design of these two modes in Section 3.3.

	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	0.20	506.10	0.11	(10m)	(6G)
	VC for intersection	0.18	505.55	0.10	(10m)	(6G)
VC for aboveMin	1. assertion	0.28	(10m)	0.16	(10m)	(6G)
	2. assertion	(10m)	0.42	(10m)	1.09	(6G)
	3. assertion	(10m)	0.86	(10m)	(10m)	(6G)
	4. type invariant	0.64	(10m)	0.31	(10m)	(6G)
	5. postcondition	0.02	0.03	0.05	0.01	0.00
VC for aboveMin	1. precondition	0.01	0.05	0.07	0.01	0.01
	2. precondition	0.02	0.05	0.08	0.01	0.01
	3. assertion	0.45	0.25	0.22	0.48	(6G)
	4. postcondition	1.70	0.26	0.27	(10m)	(6G)
	5. postcondition	(10m)	0.06	0.15	(6G)	(6G)
		0.43	0.31	0.26	466.08	(6G)

Fig. 3. maxUnion: proof results

```
(* core of the bit vector theory *)
constant size : int
axiom size_pos : size > 0
type t
function nth t int : bool
axiom nth_out_of_range:
  forall x:t, n:int. (n < 0 ∨ n ≥ size) → nth x n = False
constant zeros : t
axiom zeros_spec: forall n:int. nth zeros n = False
constant ones : t
axiom ones_spec: forall n:int. 0 ≤ n < size → nth ones n = True

(* bit-wise Boolean operators, shifts *)
function bw_and t t : t                  (* bit-wise 'and' of two bit vectors *)
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)
(* ... similar declarations and axioms for bw_or, bw_xor, bw_not ... *)
function lsr t int : t                              (* logical shift right    *)
function asr t int : t                              (* arithmetic shift right *)
function lsl t int : t                              (* logical shift left     *)
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
(* ... similar axioms for lsr and asr ... *)
```

Fig. 4. Generic theory for bit vectors: core, bit-wise Boolean operators and shifts

3 The Why3 Bit Vector Theory

Our theory of bit vectors is generic with respect to the size of bit vectors. It is then instantiated for size 8, 16, 32 and 64. In Why3, such an instance is possible through the so-called *cloning* feature: when a theory has one or more components that are declared abstract (a type, a function symbol) then one can *clone* that theory while giving some instance to some or all of these abstract components. This results in a new theory containing a copy of the original theory, with all declarations appropriately instantiated.

In the following, we only describe a representative part of the theory. We refer to the report [14] for its full description as well as a discussion of its consistency and soundness, which is established through realizations in the Coq proof assistant and in Isabelle/HOL as well.

3.1 Bit-Wise Operators

The first part of the theory is shown in Figure 4. It starts with the declaration of the (positive) parameter `size`, representing the number of bits of all bit vectors. The type of

```
constant two_power_size : int = pow2 size
constant max_int : int = two_power_size - 1
function to_uint t : int                 (* conversion to an unsigned integer *)
function of_int int : t                  (* conversion from any integer
                                            (taken modulo two_power_size)     *)
constant size_bv : t = of_int size       (* bit vectors size, as a bit vector *)
axiom Of_int_zeros : zeros = of_int 0
axiom Of_int_ones: ones = of_int max_int
axiom to_uint_extensionality : forall v,v':t. to_uint v = to_uint v' → v = v'
predicate uint_in_range (i : int) = 0 ≤ i ≤ max_int
axiom to_uint_bounds : forall v:t. uint_in_range (to_uint v)
axiom to_uint_of_int : forall i:int. uint_in_range i → to_uint (of_int i) = i

predicate ult (x y:t) = to_uint x < to_uint y         (* unsigned 'less than' *)
(* ... similar def for ule, ugt, uge ...  *)
function add t t : t                                              (* addition *)
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 x y:t. to_uint x + to_uint y < two_power_size →
  to_uint (add x y) = to_uint x + to_uint y
(* ... similar declarations for sub, neg, mul, udiv, urem ...  *)
```

Fig. 5. Bit Vector theory: conversions and arithmetic

bit vectors is introduced as an abstract type `t` equipped with one uninterpreted function `nth`. The intended meaning is that (`nth` b n) gives the n-th bit of b, as a Boolean. Note the convention that bit 0 is the least significant bit, and (`nth` b n) returns `False` when n is out of the range $0 \ldots \texttt{size} - 1$. We introduce two constants `zeros` and `ones` for the bit vectors that have all bits not set or set, respectively. These are axiomatized using `nth`.

The bit-wise operators 'and', 'or', 'xor' and 'not' come next. Their behavior is axiomatized with the help of the `nth` operator as seen in Figure 4. Shift operators are also axiomatized using the `nth` operator. Notice that the second argument of shift operators is an integer and not a bit vector.

3.2 Conversion To and From Integers

The second part of our theory, presented in Figure 5, deals with conversion between bit vectors and integers. For lack of space, we only describe here the interpretation of bit vectors as non-negative integers, that interprets $b_{n-1} \cdots b_1 b_0$ as $\sum_{i=0}^{n-1} b_i \times 2^i$. We start by defining the maximum representable integer, and its successor: 2 to the power of `size`. Then we introduce two abstract functions for the conversions. These are not fully specified from `nth`; it would be a very involved axiomatization that is unlikely to be useful for automated provers. Instead, we provide a few useful axioms on those functions, regarding constants `size`, `zeros` and `ones`, and relation to equality.

```
function nth_bv t t : bool                    (* same as nth with bv arguments *)
axiom nth_bv_def:
  forall x i:t. 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)
axiom Nth_is_nth_bv: forall x:t, i:int. uint_in_range i →
  nth_bv x (of_int i) = nth x i
function lsr_bv t t : t                       (* same as lsr with bv arguments *)
axiom lsr_bv_is_lsr: forall x n:t. lsr_bv x n = lsr x (to_uint n)
axiom to_uint_lsr: forall v n:t.
  to_uint (lsr_bv v n) = div (to_uint v) (pow2 ( to_uint n ))
(* ... similar def and axioms for lsl_bv and asr_bv ... *)

predicate eq_sub (a b:t) (i n:int) =              (* a[i..i+n-1] = b[i..i+n-1] *)
  forall j:int. i ≤ j < i + n → nth a j = nth b j
predicate eq_sub_bv (a b:t) (i n:t) =      (* same as eq_sub with bv arguments *)
  let mask = lsl_bv (sub (lsl_bv (of_int 1) n) (of_int 1)) i (* ((1<<n)-1)<<i *)
  in bw_and b mask = bw_and a mask                      (* a & mask = b & mask *)
axiom 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
```

Fig. 6. Additional operators in the bit vector theory

Arithmetic operations do not need to distinguish between signed and unsigned variants, except for division and remainder. Their behavior is axiomatized via `to_uint` to express that computation is done modulo $2^{\texttt{size}}$. Derived lemmas like `add_bounded` are added to help provers.

3.3 Strategy for Isolating Bit-Level Reasoning

The set of operators that we defined so far is expressive enough to formally specify programs. In order to discharge VCs a first idea would be to map each symbol of our theory to the corresponding symbol in the SMT-LIB theory, provided such a symbol exists, whilst keeping the other symbols uninterpreted and keeping all the axioms. However, we observed that this is not sufficient in practice: provers do not work well on VCs mixing bit-wise operators and conversions with integers (provers with native support for bit vectors have a hard time mixing bit vectors and integers, provers without it have a hard time to reason on bit-wise operators with the axioms only). Our approach to overcome this issue is two-fold. First, we provide a means for the user to isolate pure bit vector VCs from other VCs. Second, we provide to provers two alternative translations of our bit vector theory, to target specifically either provers with native support, or provers without it. The proof strategy used for the Rightmost Bit trick example (Figure 3) exploits this approach.

Bit-level operator variant. The theory is augmented with the additional operators presented in Figure 6. We provide pure bit vector alternatives for `nth` and shifts.We also introduce the `eq_sub` operator and its bit-level variant `eq_sub_bv`.

object	prover with native BV support	prover without native BV support
type t	(_ BitVec 32)	abstract
nth	uninterpreted	uninterpreted
zeros	#x00000000	uninterpreted
ones	#xFFFFFFFF	uninterpreted
bw_and	bvand	uninterpreted
axioms {zeros,ones,bw_and}_spec	removed	kept
add	bvadd	uninterpreted
axiom add_spec	removed	removed
lemma add_bounded	removed	kept
lsr	uninterpreted	uninterpreted
axioms lsr_spec_{low,high}	removed	kept
to_uint	bv2nat	kept
axiom to_uint_extensionality	removed	kept
of_int	nat2bv	kept
nth_bv	uninterpreted	uninterpreted
axiom nth_bv_def	kept	removed
axiom nth_bv_is_nth	kept	kept
lsr_bv	bvlshr	uninterpreted
axiom lsr_bv_is_lsr	kept	kept
eq_sub	uninterpreted	uninterpreted
eq_sub_bv	uninterpreted	uninterpreted
axiom eq_sub_bv_def	kept	removed

Fig. 7. Mapping to SMT-LIB, for the case `size`=32

The two drivers. Why3's *driver* mechanism allows us to tell for each object (type, function symbol) of the Why3 theory what is the syntax for the corresponding object of the target prover. Figure 7 summarizes the two driver variants for the instance of the theory with `size`=32. The second column is the mapping for provers with native bit vector support, the third column is for the other provers as well as for the noBV variants of CVC4 and Z3. The driver for provers with native support maps the type t to the corresponding type in SMT-LIB. Each operator is mapped to the corresponding symbol in the SMT-LIB theory, if it exists, and is kept uninterpreted otherwise. The axioms that link the uninterpreted operators with the native ones are kept as-is. The remaining axioms are removed. There are two exceptions: `nth_bv` and `eq_sub_bv` are not in the SMT-LIB theory. Therefore, we keep the axioms that define them in term of pure bit-level operators. The driver for provers without native support keeps all symbols uninterpreted. All the axioms are kept except the ones that define the bit-wise operators, in order to prevent the provers from trying to prove bit-level properties.

4 Adding Support for Bit Vectors in SPARK2014

Ada 2012 is the latest version of the Ada language [1], a programming language targeting real-time embedded software that requires a high level of safety, security, and

reliability. This version adds new features for specifying the behavior of programs, such as subprogram contracts and type invariants. SPARK is a subset of Ada targeting formal verification [11,21]. Its restrictions ensure that the behavior of a SPARK program is unambiguously defined. 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. It provides dedicated features that are not part of Ada 2012. Essential constructs for formal verification (*e.g.* loop invariants) have also been introduced. To formally prove a SPARK 2014 program, GNATprove uses the language WhyML as an intermediate. The SPARK program is translated into a WhyML program which can then be verified using the Why3 tool.

Modular Integer Types. Ada's very rich type system allows us to define various kinds of integer types. There are mostly of two kinds, namely *signed* and *modular* integer types. Modular integer types are defined by specifying a modulus, and are the types on which bit-wise operations apply. For example

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

defines a type `BV8` that contains unsigned integers between 0 and $2^8 - 1$. Overflows never occur when computing with it: computations use modular arithmetic semantics. The package `Interfaces` from Ada's standard library introduces predefined names `Unsigned_8`, `Unsigned_16`, `Unsigned_32` and `Unsigned_64`, respectively for the modular types modulo 2^8, 2^{16}, 2^{32} and 2^{64}. Bit-wise Boolean operations are written as infix operators `and`, `or`, `xor`, `not`. Ada provides, in its standard library, functions `Shift_Left`, `Shift_Right`, and `Shift_Right_Arithmetic`. These are defined only when the first argument is a modular type for the standard bit sizes 8, 16, 32, and 64. The second argument of these operations is not of modular type but of type `Natural`, that is the signed integer type of only non negative values defined in Ada's standard library.

Handling of Modular Types in SPARK 2014. GNATprove translates each Ada variable, resp. each expression, into a Why3 variable, resp. expression, of some adequate type [17]. Variables and expressions of some modular type are translated into variables and expressions of some bit vector type of the Why3 theory described in the previous section. Their size is either 8, 16, 32, or 64, the smallest of those that can represent all the values of the original Ada type. To simplify the presentation below, we consider only the four predefined modular types `Unsigned_8`, `Unsigned_16`, `Unsigned_32` and `Unsigned_64` corresponding to 8, 16, 32, and 64-bits integers. The translation of the Boolean bit-wise operations is directly the equivalent introduced in our Why3 theory. The translation of shifts is just slightly more complex because their second argument in Ada is a signed type and not a modular type. For instance, we translate `Shift_Left(X,Y)` as `(lsl_bv X (if Y < size then (of_int Y) else size_bv))`.

5 The "Bitwalker" Case Study, Using SPARK2014

The original C version of the BitWalker was provided by Siemens in the context of the ITEA 2 project OpenETCS. The version presented here was rewritten by Fraunhofer FOKUS to simplify the formal verification with Frama-C/WP [16]. The formal

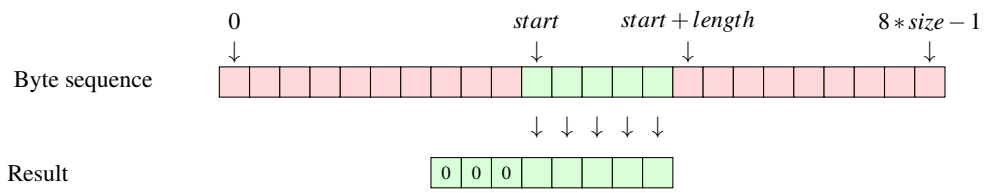

Fig. 8. Schematic view of the Peek function (on 8-bit instead of 64-bit)

```
// sets the bit at index [left] in [value] to the value of [flag]
static inline 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);
}

// return the 64-bit value extracted from the byte sequence [addr],
// from index [start] to index [start+length-1]
uint64_t 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;
}
```

Fig. 9. The BitWalker, C version, the Peek function

specification relies on a theory of bit vectors designed in the Coq proof assistant, and a significant part of the proofs were done interactively within Coq.

Bitwalker is about interacting with a stream of bytes. One of the two main functions, `Peek`, copies a value from the byte stream to a 64-bit unsigned integer. The expected behavior of `Peek`, illustrated in Figure 8, can be expressed at a high-level by saying that the integer value of the result is the value read in the byte stream starting from the bit number `start` and reading `length` bits. The most significant bits of the result, of index larger or equal to `length`, must be all zero. Figure 9 presents the C source code of `Peek` as well as one of its main auxiliary function, `PokeBit64`. The code of `Peek` does not make use of low-level bit-wise operators, but calls instead auxiliary functions. On the contrary, the code of low-level auxiliary functions `PeekBit8` and `PokeBit64` make use of bit-wise operators, so there is a need at some point to relate those bit-wise operations with more high-level arithmetic notions. In the following, we propose a SPARK program equivalent to the C code of Figure 9, with appropriate formal specifications.

Specification and Verification of `PokeBit64` The function `PokeBit64` writes a bit in an `Unsigned_64` value at the given position `Left`. In order to specify this we need to: first

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

Fig. 10. Specifications of auxiliary functions for `Peek`

write that the mentioned bit is correctly set after the function is called, and then not to forget that all other bits remain unchanged. Its SPARK specification is given in Figure 10. A first difference between the C and SPARK version appears in the types: in C, the first two parameters are unsigned types and the third parameter is an integer. In Ada, since the function manipulates the first parameter's bits, it has to be of modular type. However, the parameter `Left` represents a position: it is not intended to be manipulated at the level of its bits and we do not want a modular arithmetic semantics, hence we set its type to `Natural`. This is consistent with the typing of shifts in Ada as described in Section 4. The last parameter, as it represents the state of a bit, is naturally given the type `Boolean`. Note the use of function `Nth` which refers to the Why3 operator `nth`. While the SPARK language does not have this function built in, we use the SPARK feature *external axiomatization* to lift it, as well as some others, to the level of SPARK language [14].

The verification of `PokeBit64` is not straightforward: we are in the case of a mix of bit vectors and integers. Following the proof strategy of Section 3.3 we introduce assertions to separate the part dischargeable by provers with native bit vector support from the rest. The code, with the assertions used to prove the specification, is given in Figure 11. The third and last assertions reformulate the postcondition for CVC4 and Z3 at the bit level. The three other assertions deal with conversions between modulars and integers, and are proved by other provers.

Specification and Proof of Bitwalker `Peek`. The SPARK specification of the main function `Peek` is given in Figure 12. As for `PokeBit64` there is a difference in the types: in Ada, `Start` and `Length` are naturals, by extension to what was said on `PokeBit64` type. Note also the absence of the parameter `size`: it corresponds to `Addr'Length` in Ada. The precondition starts on line 13, by specifying that the first index of our byte sequence is 0, as in the C code. We then bound `Length`, the number of bits to copy, by 64. The last two preconditions are here to avoid any arithmetic overflow with `Start`, `Length`, and the size of `Addr`. The postcondition starts on line 17, and is made of two disjoint cases. First, if the last bit to copy is out of the bounds of the byte sequence the default value 0 is returned. In the other case, we specify two things: that the i-th bit of the result, for $0 \leq i < \texttt{Length}$ is equal to the bit of the sequence at position $\texttt{Start} + \texttt{Length} - i - 1$, as shown in Figure 8. The n-th bit of a `ByteSequence` is specified by the auxiliary function `Nth8_Stream` given on line 3 of Figure 12. Finally we specify that the other bits of the result are set to zero.

```
function PokeBit64(Value: Unsigned_64; Left: Natural;
                   Flag: Boolean) return Unsigned_64 is
 Left_Bv: constant Unsigned_64 := Unsigned_64(Left);
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 obligations	Alt-Ergo (1.01)	CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.2)	Z3 (4.4.2 noBV)
1. assertion	0.05	(10m)	0.08	0.33	7.95
2. precondition	0.03	0.11	0.08	0.03	0.14
3. assertion	0.29	(10m)	0.07	0.14	(6G)
4. precondition	0.04	0.14	0.05	0.02	0.11
5. range check	0.03	0.05	0.04	0.01	0.01
6. range check	0.03	0.04	0.04	0.01	0.00
7. assertion	(10m)	0.44	(10m)	0.21	(6G)
8. assertion	0.36	(10m)	0.10	0.23	(6G)
9. range check	0.06	0.03	0.04	0.01	0.00
10. assertion	(10m)	0.15	(10m)	0.10	(6G)
11. precondition	0.08	0.04	0.02	0.01	0.01
12. range check	0.04	0.04	0.04	0.00	0.01
13. range check	0.05	0.03	0.04	0.01	0.00
14. range check	0.04	0.03	0.04	0.01	0.00
15. postcondition	(10m)	0.23	0.11	(10m)	(6G)

Fig. 11. PokeBit64: annotated code and proof results

The Ada code of `Peek` is very close to the original C code of Figure 9. We only add two loop invariants (lines 12-18) that are directly derived from the post-conditions. These invariants are the expected ones in presence of such a loop. Note that, following our reasoning on type assignment, `Start` and `Length` are Naturals, whereas the contents of the array `Addr` are 8-bit modular types, and the result of `Peek` is a 64-bit modular. As expected, since there is no bit-level code in `Peek`, there is no need for bit-level assertions and the proof does not need the provers with native bit vector support.

6 Conclusions

We designed a rich formal theory including arbitrary fixed-size bit vectors, a large set of bit-wise operations, and a large set of operations involving both bit vectors and unbounded integers. Thanks to the driver mechanism of Why3, proof obligations that make use of this theory can be discharged either by SMT solvers with bit vector support (CVC4, Z3) or by solvers that handle this theory as an axiomatic first-order theory (Alt-Ergo, and CVC4 and Z3 in non native support mode). We presented several case studies illustrating how one can specify and prove bit-level code correct with respect to a high-level specification. We emphasize that it is important for the user to understand well the respective capabilities of the provers (do they support bit vector theories or not) and to respect a refinement-like methodology when writing annotations: to prove that bit-level code satisfies a high-level postcondition, one may need to provide a hint in the

```
type Byte_Sequence is
  array (Natural range ≠) of Unsigned_8;

function Nth8_Stream (Stream : Byte_Sequence;
                      Pos : Natural) return Boolean is
  (Nth (Stream (Pos / 8), 7 - (Pos rem 8)))
with Pre => Stream'First = 0 and then
            (Pos / 8 ≤ Stream'Last), Ghost;

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)));
```

```
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;
```

Fig. 12. Ada specification and body of `Peek` function

form of an assertion rephrasing the postcondition at the bit-level, and help the provers with assertions to enforce them to convert bit vectors to integers when required. Fortunately, as shown by proof of `Peek` in BitWalker, our approach allows a good modularity principle: as soon as low-level code is given a high-level specification, the procedures calling such code do not need to be aware that the low-level code operates at the bit level. The support of Ada's modular types via bit vectors is included since 2015 in SPARK releases. The first feedback from AdaCore's customers is very positive: many proof obligations that were not checked automatically before are now proved by CVC4 or Z3.

About SPARK interpretation of signed integers. We chose to map Ada's signed integer types to mathematical unbounded integers. Another choice would be to map them to bit vectors and use the signed arithmetic operators provided by SMT-LIB. We tried this alternative and noticed regressions in the rate of automatically proved VCs: on the SPARK test suite the support for unbounded integer arithmetic in SMT solvers is better than the support for arithmetic operators of BV theory.

Related tools and experiments. Stefan Berghofer (Secunet, Germany) is using the support for bit vectors in SPARK, on the big number package of libsparkcrypto (https://bitbucket.org/sberghofer/libsparkcrypto/). He uses Isabelle/HOL to interactively discharge the VCs that cannot be proved automatically. The BitWalker case study was initially written in C and specified using the ACSL specification language of Frama-C. For that purpose a theory of bit vectors of unbounded size was designed using the Coq proof assistant, and the proofs were done with a significant amount of interaction within Coq. Thanks to the mapping of our bit vector theory to SMT-LIB we were able to prove BitWalker fully automatically. The source language, C or Ada, is not important, although the choice between signed versus unsigned types in the source makes a difference: in Ada their semantics are significantly different. The Boogie [2] verifier and its front-ends VCC [13] and Dafny [19] also use the built-in bit vector support of Z3, to model machine words. We are not aware of any work, in this context, about the problem of mixing bit vectors with high-level specifications.

Future Work. The need to use two different drivers for the same prover is somehow unsatisfactory. The decision of using the native support for bit vectors in provers could be made by an automatic analysis of the goal. A possible alternative would be to provide appropriate constructs in the specification language so that the user could indicate the intended level of abstraction in her code. For instance, in our solution to the n-queens example [14], it would have been convenient to express with a source annotation that we want to interpret a machine word into the set of positions of its bits set to 1.

There is some need to apply the same approach to floating-point numbers, in order to exploit decision procedures for floating-point arithmetic that are now available in SMT solvers [9] (http://www.cprover.org/SMT-LIB-Float/). In the past, floating-point programs were specified in terms of real numbers [8] and proved by specific solvers. As we did for bit vectors and integers, it is therefore desirable to design a theory that would allow the combination of floating-point numbers with real numbers and at the same time would make use of SMT-LIB support for floating-point arithmetic. Last but not least, there are some programs that operate on floating-point numbers at the bit-level [20]. Proving such code would be a hard challenge [23].

Acknowledgments. Thanks to Stefan Gerken from Siemens for providing the original implementation of the BitWalker. Thanks to Stefan Berghofer for providing us with an Isabelle/HOL realization of Why3's bit vector theory. Thanks to Jean-Christophe Filliâtre, Stuart Matthews, Yannick Moy and Mário Pereira for their comments on preliminary versions of this paper.

References

1. Barnes, J.: Programming in Ada 2012. Cambridge University Press (2014)
2. Barnett, M., DeLine, R., Jacobs, B., Chang, B.Y.E., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: FMCO. LNCS, vol. 4111, pp. 364–387 (2005)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177. LNCS, Springer (2011)

4. Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: Design Automation Conference. pp. 522–527. ACM (1998)
5. Berry, G.: The foundations of Esterel. In: Proof, Language, and Interaction, Essays in Honour of Robin Milner. pp. 425–454. The MIT Press (2000)
6. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008), http://alt-ergo.lri.fr/
7. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. Int. Journal on Software Tools for Technology Transfer 17(6), 709–727 (2015)
8. Boldo, S., Marché, C.: Formal verification of numerical programs: from C annotated programs to mechanical proofs. Mathematics in Computer Science 5, 377–393 (2011)
9. Brain, M., Tinelli, C., Ruemmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: ARITH. pp. 160–167 (2015)
10. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: An abstraction-based decision procedure for bit-vector arithmetic. Int. Journal on Software Tools for Technology Transfer 11(2), 95–104 (2009)
11. Chapman, R., Schanda, F.: Are we there yet? 20 years of industrial theorem proving with SPARK. In: ITP. LNCS, vol. 8558, pp. 17–26. Springer (2014)
12. Cyrluk, D., Rueß, H., Möller, O.: An efficient decision procedure for the theory of fixed-sized bit-vectors. In: Computer Aided Verification. vol. 1254, pp. 60–71. Springer (1997)
13. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C. In: ICSE. pp. 429–430. IEEE Comp. Soc. Press (2009)
14. Dross, C., Fumex, C., Gerlach, J., Marché, C.: High-level functional properties of bit-level programs: Formal specifications and automated proofs. Research Report 8821, Inria (2015)
15. Filliâtre, J.C.: Verifying two lines of C with Why3: an exercise in program verification. In: VSTTE. LNCS, vol. 7152, pp. 83–97. Springer (2012)
16. Gerlach, J.: Validation and verification of implementation/code. Tech. Rep. D4.3.2, OpenETCS (2015), https://github.com/openETCS/governance/wiki/State-of-Deliverables
17. Kanig, J., Schonberg, E., Dross, C.: Hi-Lite: the convergence of compiler technology and program verification. In: HILT. pp. 27–34. ACM Press (2012)
18. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Experience report. In: FM. LNCS, vol. 6664. Springer (2011)
19. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: F-IDE. EPTCS, vol. 149, pp. 3–15 (2014)
20. Lomont, C.: Fast inverse square root. Tech. rep., Indiana: Purdue University (2003), http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf
21. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
22. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
23. Nguyen, T.M.T.: Taking architecture and compiler into account in formal proofs of numerical programs. Thèse de doctorat, Université Paris-Sud (2012)
24. Warren, H.S.: Hackers's Delight. Addison-Wesley (2003)