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

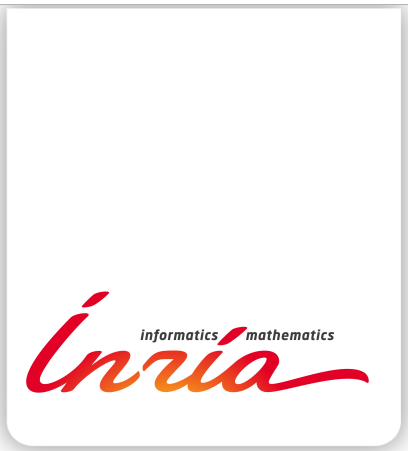
Claire Dross, Clément Fumex, Jens Gerlach, Claude Marché

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

Claire Dross, Clément Fumex, Jens Gerlach, Claude Marché

RESEARCH REPORT
N° 8821
December 2015
Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--8821--FR+ENG



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

Claire Dross*, Clément Fumex†, Jens Gerlach‡, Claude Marché†§

Project-Team Toccata

Research Report n° 8821 — December 2015 — 52 pages

Abstract: In a computer program, basic functionalities may be implemented using bit-wise operations. This can be motivated by the need to be close to the underlying architecture, or the need of efficiency, both in term of time and memory space. If one wants 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 a low-level code conforms to a higher-level specification is challenging, because of the gap between the different levels of abstraction.

Our approach to address this challenge is to design 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 experimentally.

Key-words: Formal specification, deductive verification, formal proof, fixed-size bit-vectors

* AdaCore, F-75009 Paris
† Inria, Université Paris-Saclay, F-91893 Palaiseau
‡ Fraunhofer FOKUS, Berlin, Germany
§ LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Propriétés fonctionnelles de haut-niveau pour des programmes opérant au niveau des bits: spécifications formelles et preuves automatiques

Résumé : Dans un programme informatique, des fonctionalités de base sont parfois implémentées par des opérations bit-à-bit, par exemple à cause d'un besoin d'être proche de l'architecture matérielle sous-jacente, ou bien pour des questions d'efficacité, aussi bien en temps de calcul qu'en place mémoire. Si l'on cherche à spécifier formellement le comportement attendu d'un tel programme, il est souhaitable que la spécification se place à un niveau plus abstrait que celui des bits. Prouver formellement qu'un programme bas niveau est conforme à une spécification de plus haut niveau est un défi, à cause de l'écart important entre les niveaux d'abstraction en jeu.

Notre approche pour résoudre ce défi consiste à concevoir une théorie formelle des vecteurs de bits, qui d'une part permet à un utilisateur d'écrire des spécifications proches d'un niveau de conception humain (ou bien disons, mathématique), et d'autre part peut se connecter aux procédures de décision et aux outils sachant traiter les vecteurs de bits, comme ceux qui sont développés dans le cadre SMT (*Satisfiability Modulo Theory*).

Cette approche est implémentée dans le cadre de l'environnement généraliste Why3 pour la preuve de programme, ainsi que dans l'environnement SPARK pour le développement de codes critiques en Ada, qui utilise Why3 en interne. Nous présentons plusieurs études de cas afin de valider notre approche.

Mots-clés : Spécification formelle, preuve de programmes, vecteurs de bits

Contents

1 Introduction

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

As an extreme example we can mention the following 2-lines C program, a so-called "signature program", designed Marcel van Kervinc[1].

```
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));}
```

This program reads an integer n from the standard input and prints another integer $f(n)$ to the standard output. 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 a code a formal specification, 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 such a specification, by theorem proving. The solutions presented in the past by Filliâtre [14] and other authors [18] considered a more abstract implementation, 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 all are tautologies, then the program is guaranteed to respect its specification. In recent 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 [5], CVC4 [2] and Z3 [13]. 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 a quite long time, and efficient decision procedures are known [12, 3, 9]. The SMTLIB international initiative[2] aims at providing standard languages and descriptions of theories for interacting with SMT solvers. SMTLIB provides a fairly rich standard theory for fixed-size bit-vectors, and this theory is implemented in several SMT solvers, including CVC4 and Z3.

Our objective is to add a support for bit-wise operations in Why3 and its front-end SPARK2014 [10] dealing with safety-critical Ada programs. For this purpose, we want to exploit the bit-vectors decision procedures provided by SMT solvers. However, in such a context, bit-wise operations are mixed with other objects occurring in programs and in specifications, such as unbounded integers, arrays, records. We thus need to rely on other theories supported by SMT solvers, in particular integer and real arithmetic, and functional arrays. Exploiting the SMT solver when all those theories are mixed together requires special care. This paper reports on our design choices and on some experiments we made. In Section 2, we first present the theories for bit-vectors we designed for use in the Why3 program verifier. In Section 3, we present two case studies on short smart programs: an efficient program to count the number of bits in a bit-vector, and a proof of the code given above that solves the n-queens using bit-wise operations. In Section 4, we present how our Why3 theories for bit-vectors are exploited in the SPARK2014 front-end, and we illustrate our approach on a case study originating from an industrial code. These developments will be distributed in the future releases of Why3 (version 0.87) and SPARK (SPARK Pro 16.0). The files for the case studies are available on Toccata's Web gallery of verified programs[3].

[1] `http://www.iwriteiam.nl/SigProgC.html`

[2] `http://smtlib.cs.uiowa.edu/`

[3] `http://toccata.lri.fr/gallery/bitwise.en.html`

2 A Theory Mixing Bit-Vectors and Integer Arithmetic

Our objective is to specify programs, that directly manipulate bits, at an abstract level. Our approach is to exploit in parallel the theories of unbounded integers, functional arrays and such ; and the theory of bit-vectors supported by SMT-solvers. We want to provide a theory that let the user use 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.

As a running example of this section let's consider the trick proposed in the book Hacker's Delight [24, page 12] to find the position of the rightmost bit set in a given bitvector `x`, using the expression `x & (-x)`. The result is a bitvector with exactly one bit set which corresponds to the rightmost bit set of `x`. For example, the rightmost bit set of `x=10101100` is given by `10101100 & (-10101100)` = `10101100 & 01010100` = `00000100`. Note that this trick is used in the 2-lines C code shown in the introduction (somehow hidden in the expression `(e-=d)&-e`).

We would like to formally specify this expected behavior at an abstract level, in particular not mentioning any low level operation like a bit-wise 'and'. Let's assume we work on bit-vectors of size 32, and that the bits are indexed from 0 (rightmost bit) to 31 (leftmost bit). We want to formally express the following informal behavior: there exists a bit position p such that

- p is between 0 and 31
- each bit of x at a position smaller than p is not set
- the bit of x at position p is set
- each bit of the result is not set except the one at position p

In this section we start with a quick introduction to the Why3 environment, then we present our bit-vector theories, and finally we come back to our solution to this running example.

2.1 Why3 in a Nutshell

Why3 is an environment for deductive program verification, providing a rich language for specification and programming, called WhyML. It relies on external provers, both automated and interactive, in order to discharge the auxiliary lemmas and verification conditions. WhyML is used as an intermediate language for verification of C, Java or Ada programs [15], and is also intended to be comfortable as a primary programming language.

The specification component of WhyML [6], used to write program annotations and background theories, is an extension of first-order logic. It features ML-style polymorphic types (prenex polymorphism), algebraic datatypes, inductive and co-inductive predicates, recursive definitions over algebraic types. Why3 comes with a rich standard library providing general-purpose theories useful for specifying programs [7]. This naturally includes integer and real arithmetic.

The user can formalize its own additional theories. A new type, a new function, or a new predicate can be either defined or declared abstract and axiomatized. For example, one can formalize a theory of sets of integers as follows

```
theory IntSet
  use import int.Int              (* theory requires integer arithmetic for Why3's stdlib *)
  type t                          (* abstract type of sets of integers *)
  predicate mem int t             (* (mem x s) means ``x in t'' *)
```

```
  function empty : t                (* empty set *)
  axiom mem_empty: forall x:int. not (mem x empty)
  function single int : t           (* (single x) is the singleton set {x} *)
  axiom mem_single: forall x y:int. mem x (single y) ↔ x=y
  function union t t : t            (* union of two sets *)
  axiom mem_union:
    forall x:int, s1 s2:t. mem x (union s1 s2) ↔ (mem x s1)  ∨  (mem x s2)
  function incr_all t : t           (* increment by 1 all elements in a set *)
  axiom incr_spec : forall x:int, s:t. mem x (incr_all s) ↔ mem (x-1) s
end
```

The specification part of the language can serve as a common format for theorem proving problems, suitable for multiple provers. The Why3 tool generates proof obligations from lemmas and goals, then dispatches them to multiple provers, including SMT solvers Alt-Ergo, CVC4, Z3; TPTP first-order provers E, SPASS, Vampire; interactive theorem provers Coq, Isabelle and PVS. For example, one can state some lemma on sets of integers:

```
theory Test
  use IntSet
  lemma toy_test : mem 42 (incr_all (union (single 17) (single 41)))
end
```

Such a lemma is easily proved valid by SMT solvers, since they are able to handle formulas mixing integer arithmetic, equality and uninterpreted symbols axiomatized by first-order axioms.

As most of the provers do not support some of the language features, typically pattern matching, polymorphic types, or recursion, Why3 applies a series of encoding transformations to eliminate unsupported constructions before dispatching a proof obligation. Other transformations can also be imposed by the user in order to simplify the proof search.

The programming part of WhyML is a dialect of ML with a number of restrictions to make automated proving viable. WhyML function definitions are annotated with pre- and post-conditions both for normal and exceptional termination, and loops are also annotated with invariants. In order to ensure termination, recursive definitions and while-loops can be supplied with variants, *i.e.* values decreasing at each iteration according to a well-founded order. Statically-checked assertions can also be inserted at arbitrary points in the program. WhyML also features *ghost* code, computations that only help verification and can be safely removed from a program without affecting its visible behavior.

The Why3 tool generates proof obligations, called verification conditions, from those annotations using a standard weakest-precondition procedure. Also, most of the elements defined in the specification part: types, functions, and predicates, can be used inside a WhyML program. We refer to Filliâtre and Paskevich [16] for more details on the programming part of WhyML. Why3's web site[4] also provides a extensive tutorial and a large collection of examples. The WhyML source code in this paper is mostly self-explanatory for those familiar with functional programming.

2.2 The Why3 Bit-Vector Theory

The theory we present here is generic with respect to the size of bitvectors. This generic theory 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, or a predicate) then one can *clone* that theory while giving some instance to some or all these abstract components. This results in a new theory containing a copy of the original theory, with all axioms appropriately instantiated.

[4] `http://why3.lri.fr`

The four instances of this generic theory must be handled appropriately by Why3 when the verification conditions are passed to SMT solvers: some mechanism should tell for each object (type, function symbol) of the Why3 theory what is the syntax for the corresponding object of the target solver. In Why3, this is done using a so-called *driver*, a text file that precisely defines this correspondance. On each of the components below, we will detail how we chose to translate it in the SMTLIB driver of Why3.

2.2.1 Core Bit-Vector Theory

The core of our generic theory starts with the declaration of the (positive) parameter `size`, representing the number of bits of all bit-vectors:

```
constant size : int
axiom size_pos : size > 0
```

Then, the type of bit-vectors is introduced as an abstract type `t` on which there is only one function `nth` operating on it. The intended meaning is such 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 $0 \ldots$ `size` $- 1$.

```
type t
function nth t int : bool
axiom nth_out_of_range: forall x n. n < 0  ∨  n >= size → nth x n = False
```

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`:

```
constant zeros : t
axiom zeros_spec: forall n:int. nth zeros n = False
constant ones : t
axiom ones_spec: forall n. 0 <= n < size → nth ones n = True
```

Mapping to SMTLIB. In the Why3 driver for SMTLIB, we declare rules to map the objects of our theory to objects known by SMT solvers. This is done for each clones. Let's consider the clone for `size`=32. The abstract type `t` is mapped to the SMTLIB fixed-size bit-vectors type `(_ BitVec 32)`. We map as well `zeros` to `#x00000000` and `ones` to `#xFFFFFFFF`. On the other hand, the function `nth` is *not* mapped to any symbol (i.e. it remains uninterpreted) since it is not present in SMTLIB bit-vector theory, which considers instead that all bit-wise operations are primitive in this theory. Notice that the driver mechanism also allows us to remove the axioms specifying `zeros` and `ones`, so that SMT solvers supporting bit-vectors theory are not "polluted" with unnecessary extras axioms.

2.2.2 Bit-Wise Boolean Operators

The bit-wise operators 'and', 'or', 'xor' and 'not' come next in our theory. Their behavior is axiomatized with the help of the `nth` operator.

```
function bw_and t t : t     (* bit-wise 'and' of two bit-vectors *)
function bw_or t t : t      (* bit-wise 'or'                     *)
function bw_xor t t : t     (* bit-wise 'xor'                    *)
function bw_not t : t       (* bit-wise 'not'                    *)
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 axioms for or, xor and not... *)
```

Mapping to SMTLIB. These operators directly maps to the SMTLIB operators `bvand`, `bvor`, `bvxor` and `bvnot` respectively. Again, axioms that specify those operations are removed by the driver mechanism, since SMT solvers know their behavior natively.

2.2.3 Shift Operators

Similarly to the bit-wise operators, we axiomatize the shift operators using the `nth` operator. Notice that the second argument of these operations are integers and not bit-vectors.

```
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 ... *)
```

Note that these operations are specified even if the amount of the shift is bigger than size, in concordance with SMTLIB theories.

Mapping to SMTLIB. There is some gap here between our theory and the SMTLIB bit-vectors theory, since shift operators in SMTLIB take a bit-vector as second argument instead of an integer. This is why we do not map these operations to any SMTLIB bit-vector operation. We come back to this issue below in Section 2.2.8.

2.2.4 Rotation Operators

The rotation operators are dealt in the same way as shift operators: they are axiomatized with `nth` and their second argument is an integer.

```
function rotate_right t int : t

axiom Nth_rotate_right :
  forall v n i. 0 <= i < size → 0 <= n →
    nth (rotate_right v n) i = nth v (mod (i + n) size)

function rotate_left t int : t

axiom Nth_rotate_left :
  forall v n i. 0 <= i < size → 0 <= n →
    nth (rotate_left v n) i = nth v (mod (i - n) size)
```

Note that there is no restriction on the amount of the rotation.

Mapping to SMTLIB. There no rotation operator in SMTLIB bit-vectors theory. We do not map these operations to any SMTLIB bit-vector operation, although we will provide a correspondance later in Section 2.2.8.

2.2.5 Conversion To and From Integers

This part deals with conversion between bitvectors and integers. We only consider here the interpretation of bitvectors into non-negative integers[5], 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`.

```
constant two_power_size : int = pow2 size
constant max_int : int = two_power_size - 1
```

Then we introduce two abstract functions for the conversions.

```
function to_uint t : int           (* conversion to an unsigned integer *)
function of_int int : t            (* conversion from any integer
                                        (taken modulo two_power_size)   *)
```

These are not fully specified from `nth`, it would be a very involved axiomatization that is unlikely useful for automated provers. Nevertheless, we provide a few useful axioms on them, regarding constants `size`, `zeros` and `ones`, and relation to equality.

```
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. uint_in_range i → to_uint (of_int i) = i
```

Mapping to SMTLIB. SMTLIB provides two functions for conversion between bitvectors and unsigned integers, namely `bv2nat` and `nat2bv`. SMTLIB does specify that `nat2bv` is computing the result modulo 2^{size}, so it corresponds exactly to the intended meaning of our `of_int` function. We thus map `to_uint` to `bv2nat` and `of_int` to `nat2bv`. We emphasize that those conversions between bitvectors and integers are bridges between two SMTLIB core theories (i.e., bitvectors and integers), as such SMT solvers have a hard time dealing with them. Indeed, those conversions are not part of the bitvector decision procedures in the literature. We will discuss these issues with the provers in Section 2.3.2 below.

2.2.6 Comparison Operators

The comparison operators are defined using their integer counterpart.

```
predicate ult (x y:t) = to_uint x < to_uint y  (* unsigned 'less than'        *)
predicate ule (x y:t) = to_uint x < to_uint y  (* unsigned 'less or equal'    *)
predicate ugt (x y:t) = to_uint x > to_uint y  (* unsigned 'greater than'     *)
predicate uge (x y:t) = to_uint x >= to_uint y (* unsigned 'greater or equal' *)
```

[5] We also support signed integers based on the classical 2-complement representation in our implementation, but we don't use them in this report

Mapping to SMTLIB. These symbols are directly mapped to their equivalent in SMTLIB: `bvult`, `bvule`, `bvugt` and `bvuge`. Their definition above is discarded.

2.2.7 Arithmetic Operators

Arithmetic operations do not need to distinguish between signed and unsigned variant, except for division and remainder:

```
function add t t : t            (* addition           *)
function sub t t : t            (* subtraction        *)
function neg t : t              (* negation           *)
function mul t t : t            (* multiplication     *)
function udiv t t : t           (* unsigned division  *)
function urem t t : t           (* unsigned remainder *)
function sdiv t t : t           (* signed division    *)
function srem t t : t           (* signed remainder   *)
axiom add_spec: forall x y:t.
  to_uint (add x y) =  mod (to_uint x + to_uint y) two_power_size
(* ... similar axioms for other arithmetic operators ...  *)
```

Mapping to SMTLIB. These operators directly maps to the corresponding SMTLIB operators. Again, axioms that specify those operations are removed by the driver mechanism, since SMT solvers know their behavior natively.

2.2.8 Bit-Vectors Alternatives for `nth` and Shifts

As mentioned above, the shift operators in SMTLIB take a bit-vector as second argument. Indeed these variants can be useful on examples, such as the ones we will study in next sections. Here are how we axiomatize them:

```
function nth_bv t t : bool
function lsr_bv t t : t
function asr_bv t t : t
function lsl_bv t t : t
function rotate_right_bv t t : t
function rotate_left_bv  t t : t
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 i: t.
  uint_in_range i → nth_bv x (of_int i) = nth x i
axiom lsr_bv_is_lsr: forall x n. 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 axioms for lsl_bv and asr_bv ... *)
axiom rotate_left_bv_is_rotate_left :
  forall v n. rotate_left_bv v n = rotate_left v (to_uint n)
axiom rotate_right_bv_is_rotate_right :
  forall v n. rotate_right_bv v n = rotate_right v (to_uint n)
```

Mapping to SMTLIB. SMTLIB does not contain an `nth_bv` operator, however it is quite easy to write it with the ones provided, which is what we do in the driver. For example, in the case of bitvectors of length 32 bits we map a term `nth_bv x i` to the SMTLIB equivalent to the C expression `(x>>i)&1!=0`.

As for the shifts operators, we map them to the SMTLIB equivalents. The rotation operators are mapped to combination of other bitvector operation as follows: `rotate_left_bv x y` is mapped to

```
(bvor (bvshl x (bvurem y (_ bv64 64))) (bvlshr x (bvsub (_ bv64 64) (bvurem y (_ bv64 64)))))
```

and `rotate_right_bv x y` is mapped to

```
(bvor (bvlshr x (bvurem y (_ bv64 64))) (bvshl x (bvsub (_ bv64 64) (bvurem y (_ bv64 64)))))
```

2.2.9 Bit-Vectors Sub-Range Equality

Equality on bit-vectors is of course the built-in equality. However, for writing specifications we found it quite handy to introduce equality predicates for specifying that bit-vectors are equal on some sub-range.

```
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
predicate eq (v1 v2:t) = eq_sub v1 v2 0 size
axiom Extensionality: forall x y : t [eq x y]. eq x y → x = y
```

Mapping to SMTLIB. There is no counterpart of predicates `eq_sub` and `eq_sub_bv` in SMTLIB bit-vectors theory, so we simply pass these definitions to SMT solvers as they are. However, the predicate `eq` can be mapped to built-in equality, and extensionality axiom removed.

2.2.10 Instances of the Generic Theory

As mentioned in the introduction of this section, the theory of bit-vectors is written as a generic theory and is then *cloned* in 4 instances for theories of 8, 16, 32 and 64 bits bit-vectors (and could be cloned in any other size). This reflects the structure of the SMTLIB theory of bit-vectors, which is a family of theories parameterized by the size of the bit-vectors. Here is how it is done for 32-bits vectors

```
theory BV32
  constant size           : int = 32
  constant two_power_size : int = 0x1_0000_0000
  constant max_int        : int = 0xFFFF_FFFF

  clone export BV_Gen with
    constant size = size,
    constant two_power_size = two_power_size,
    constant max_int = max_int
end
```

2.2.11 Conversion Between Bit-Vectors of Different Sizes

In parallel of the four instances of the generic theory, we have another collection of theories to deal with conversions between this four instances. Again, we define first a generic theory of conversion between

two different bit-vector types, which is then cloned into six instances corresponding to the possible conversions scenarios with our four bit-vector types.

We're converting between a bigger and a smaller bitvector, so we start by declaring two corresponding abstract types.

```
type bigBV
type smallBV
```

We then declare the two functions for the conversions between these types.

```
function toBig smallBV : bigBV
function toSmall bigBV : smallBV
```

In order to specify these two functions we need a predicate that checks that a value of the bigger type is in the range of the smaller type, as well as a conversion function into integer for both types.

```
predicate in_small_range bigBV

function to_uint_small smallBV : int
function to_uint_big bigBV : int
```

We can now specify the conversions `toBig` and `toSmall` with the following axioms.

```
axiom toSmall_to_uint :
  forall x:bigBV. in_small_range x →
    to_uint_big x = to_uint_small (toSmall x)

axiom toBig_to_uint :
  forall x:smallBV.
    to_uint_small x = to_uint_big (toBig x)
```

This theory is then cloned in six instances covering the possible conversions between the four theory of bit-vectors. For example, the theory for conversions between bit-vectors of 16 bits and bit-vectors of 64 bits is defined as follows :

```
theory BVConverter_16_64
  use BV16
  use BV64

  predicate in_range (b : BV64.t) = BV64.ule b (BV64.of_int BV16.max_int)

  clone export BVConverter_Gen with
    type bigBV = BV64.t,
    type smallBV = BV16.t,
    predicate in_small_range = in_range,
    function to_uint_small = BV16.to_uint,
    function to_uint_big = BV64.to_uint
end
```

Mapping to SMTLIB. For each clone the two conversion functions are mapped, respectively, to truncation and concatenation with a zero bit-vector of appropriate size.

```
theory bv.BVConverter_16_64
  syntax function toBig "((_ zero_extend 48) %1)"
  syntax function toSmall "((_ extract 15 0) %1)"
end
```

The two axioms are removed.

2.2.12 About the Soundness of our Theory

We have designed a theory for bit-vectors of a given size. This theory is cloned for size 8, 16, 32 and 64. Another generic theory formalizes the conversions between bitvectors of different sizes, and is cloned into the six possible variants. The `bv.why` file containing the Why3 sources of these theories can be found in the Why3 distribution[6]. The Why3 driver file `smt-lib2-bv.gen` describes the mapping of these theories to SMTLIB syntax[7].

There are two questions that should be addressed to validate our design:

- Since this theory is mostly axiomatic: can we guarantee that it is consistent?
- Since this theory extends the SMTLIB bitvector theory: can we guarantee that our theory is a conservative extension, that is can we guarantee that our axioms do not contradict the assumptions of the SMTLIB bitvector theory?

To answer the first question: we build a so-called *realization* of our theory. It amounts to write an instance of our theory where there is no abstract type nor abstract symbol anymore, and where all axioms are proved valid in this instance. We thus obtain one particular model of our theory. In Why3, there is an automatic mechanism to build such a realization, using existing proof assistants. For that purpose we use the Coq proof assistant[8]. Our realization defines the type of bitvectors of size `size` as a Coq dependent type. Indeed, we where able to re-use a notion of vector already defined in Coq's standard library. All our functions are then defined to operate on this dependent type, sometimes by recursion on the size. The resulting Coq development contains 364 lines of specifications and 1174 lines of proofs, and can be found in the Why3 sources.

Regarding the second question, we can not answer in a formal way, since there is no reference formalization of the SMTLIB bit-vector theory, it is only specified informally[9]. Checking that our axioms are consistent with this theory is thus done primarily by human reading. We additionally made some checks: we turned our theory into a problem file for SMT solvers where only the function symbols that are not present in SMTLIB are declared. The axioms that defines them are turn into goals. We submitted this file to CVC4 and Z3 with a specific variant of our drivers, where only the "bridge" axioms such as `nth_bv_is_nth` are kept, hoping that the rest could be proved. Unfortunately, only the simplest of our axioms, such as the definitional axioms of bitwise operators like `Nth_bw_and` could be proved, the other still involve several conversions between bitvectors and integers, on which the SMT solvers are not very good at. A formal answer to this question remains to be found.

2.3 Specification and Proof of the Rightmost Bit Trick

2.3.1 Formal specification

We have now enough material to turn the informal specification of our running example into the following Why3 program:

```
let rightmost_bit_trick (x: t) : t
  requires { x <> zero }          (* pre-condition: x should not be zero *)
  ensures  { exists p:int.
             0 <= p < 32
           ∧ eq_sub x zero 0 p          (* all bits of x[0..p-1] are 0   *)
           ∧ nth x p                    (* bit x[p] is 1                 *)
```

[6] `https://scm.gforge.inria.fr/anonscm/gitweb?p=why3/why3.git;a=history;f=theories/bv.why`
[7] `https://scm.gforge.inria.fr/anonscm/gitweb?p=why3/why3.git;a=history;f=drivers/smt-libv2-bv.gen`
[8] Notice that meanwhile, Stefan Berghofer built another realization using Isabelle/HOL
[9] `http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml`

```
          ∧ eq_sub result zero 0 p      (* all bits of r[0..p-1] and ... *)
          ∧ eq_sub result zero (p+1) 32 (* ... r[p+1..31] are 0          *)
          ∧  nth result p }             (* bit r[p] is 1                 *)
= bw_and x (neg x)
```

There is difficulty here for proving such a code, because of the existential quantification on p. There is no chance that an automated prover could "find" p on its own. The best way to handle this in Why3 is to use an extra *ghost* parameter. The contract is modified as follows.

```
let rightmost_bit_trick (x: t) (ghost p : ref int) : t
  requires { x <> zero }
  writes   { p }
  ensures  { 0 <= !p < 32 }
  ensures  { eq_sub x zero 0 !p }
  ensures  { nth x !p }
  ensures  { eq_sub result zero 0 !p }
  ensures  { eq_sub result zero (!p+1) 32 }
  ensures  { nth result !p }
= ghost p := ??;    (* ?? needs to filled appropriately *)
  bw_and x (neg x)
```

We are left with finding an appropriate expression for p before we can proceed with the proofs.

Computing p can be done using an additional ghost function as follows.

```
  let ghost rightmost_position_set (a : t) : t
    requires { a <> zero }
    ensures  { ult result (of_int 32) }
    ensures  { eq_sub_bv a zero zero result }
    ensures  { nth_bv a result }
  =
    let i = ref zero in
    while ult !i (of_int 32) && not (nth_bv a !i) do
      variant {32 - to_uint !i}
      invariant {eq_sub_bv a zero zero !i}
      i := add !i (of_int 1)
    done;
    !i
```

2.3.2 Proof methodology

As we said before, the SMT solvers do not guarantee completeness when integers and bitvectors are mixed together. For our experiments, we proceed as follows: in one hand, we generate verification conditions for CVC4 1.4 and Z3 4.4.0 using the driver for bitvectors theory as described above. This driver maps Why3 symbols to SMT bitvector symbols when possible, and removes the axioms that become redundant. On the other hand, we generate verification conditions for Alt-Ergo 0.99.1 (which do not have any support for bit-vectors), CVC4 and Z3. We use an alternative driver, for CVC4 and Z3, that do not make use of built-in support for bit-vectors and considers instead all symbols as uninterpreted and all axioms kept. In the following, these variants of CVC4 and Z3 and called "1.4 noBV" and "4.4.0 noBV".

Therefore, we highlight two families of provers: with or without native support of bitvectors. This leads us to articulate most of our proofs in two steps : first isolate a part of the proposition to be proved that is "purely bitvector" which will be targeted to CVC4 and Z3, and then have a "converting step" which is typically targeted to Alt-Ergo, and noBV variant of CVC4 and Z3.

Following this methodology, the body of `rightmost_bit_trick` is annotated with two assertions stating two properties at the bit level:

Proof obligations		Alt-Ergo (0.99.1)	CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.0)	Z3 (4.4.0 noBV)
VC for rightmost_position_set	1. loop invariant init	3.39	0.01	0.02	0.01	(5s)
	2. loop invariant preservation	(5s)	0.03	0.07	0.02	(5s)
	3. loop variant decrease	2.15	0.06	0.02	(5s)	(5s)
	4. postcondition	0.05	0.01	0.01	0.00	0.00
	5. postcondition	0.05	0.01	0.02	0.00	0.00
	6. postcondition	0.05	0.02	0.01	0.00	0.00
	7. postcondition	(5s)	0.01	(5s)	0.01	(5s)
	8. postcondition	0.04	0.02	0.02	0.00	0.00
	9. postcondition	(5s)	0.02	(5s)	0.00	(5s)
VC for rightmost_bit_trick	1. precondition	0.04	0.00	0.02	0.00	0.00
	2. assertion	(5s)	0.02	(5s)	0.03	(5s)
	3. assertion	(5s)	0.02	(5s)	0.04	(5s)
	4. postcondition	0.20	0.02	0.03	(5s)	(5s)
	5. postcondition	0.16	0.03	0.03	(5s)	(5s)
	6. postcondition	0.02	0.02	0.02	(5s)	(5s)
	7. postcondition	(5s)	0.03	0.06	(5s)	(5s)
	8. postcondition	(5s)	0.02	0.06	(5s)	(5s)
	9. postcondition	0.40	0.03	0.03	(5s)	(5s)

Figure 1: Provers results on the Rightmost Bit Trick

```
let ghost p_bv = rightmost_position_set x in
ghost p := to_uint p_bv;
assert { nth_bv (neg x) p_bv }; (* p-th bit of -x is set *)
let res = bw_and x (neg x) in
assert {eq_sub_bv res zero (add p_bv (of_int 1)) (sub (of_int 31) p_bv )};
  (* all bits of res between positions p+1 and 31-p are not set *)
res
```

The first assertion states that the bit at position p in $-x$ is set. The second assertion states that the bits at the left of p in the result are not set.

2.3.3 Proof results

The results of the provers on the code on both the `rightmost_bit_trick` and the auxiliary ghost function `rightmost_position_set` are displayed in Figure 1. The two assertions are proved both by CVC4 and Z3, thanks to their native support for bitvectors. The other VCs are proved by the other family of provers, using these assertions and the axioms in our bit-vector theory that relate `nth_bv` and `eq_sub_bv` to their non-bv counter-parts.

```
let count (n : t) : t
  ensures { to_uint result = numof (nth n) 0 32 }
=
  let x = ref n in
  (* x = x - ( (x >> 1) & 0x55555555) *)
  x := sub !x (bw_and (lsr_bv !x (of_int 1)) (of_int 0x55555555));
  (* x = (x & 0x33333333) + ((x >> 2) & 0x33333333) *)
  x := add (bw_and !x (of_int 0x33333333))
                (bw_and (lsr_bv !x (of_int 2)) (of_int (0x33333333)));
  (* x = (x + (x >> 4)) & 0x0F0F0F0F *)
  x := bw_and (add !x (lsr_bv !x (of_int 4))) (of_int 0x0F0F0F0F);
  (* x = x + (x >> 8) *)
  x := add !x (lsr_bv !x (of_int 8));
  (* x = x + (x >> 16) *)
  x := add !x (lsr_bv !x (of_int 16));
  (* return (x & 0x0000003F) *)
  bw_and !x (of_int 0x0000003F)
```

Figure 2: Why3 code for counting bits function

3 Case studies using the Why3 Environment

3.1 Counting Bits

This is another example extracted from *Hacker's delight* [24, page 81]. It is again a smart tricky code operating at the level of bits to count how many bits are set in a given machine word. Counting the bits has several applications, such as computing the Hamming distance of two machine words, which is used in error detection and correction.

A source code in C for this trick is as follows, for 32-bits unsigned integers (`uint32_t`).

```
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) ;
}
```

The Hamming distance is then just the number of bits of the bit-wise exclusive or of two words.

```
uint32_t dHamming(uint32_t x, uint32_t y) {
  return count(x ^ y);
}
```

We would like to specify formally the high-level behavior of `count`: the returned value is the number of indices i such that the i-th bit of `x` is set, that is (`nth x` i) is true. From such a high-level specification, we should be able to derive high-level properties of `dHamming`, such as the fact it is really a distance in a mathematical point of view: symmetric, satisfies the triangle inequality.

Why3 comes with a quite rich standard library for specifying programs, in particular it provides a higher-order function `numof` such that (`numof` $p\ a\ b$) denotes the number of i, $a \leq i < b$, such that $(p\ i)$ is true. We can turn the C code of `count` into the following equivalent Why3 program shown on Figure 2.

The post-condition on line 2 reads as follows: the resulting machine word, interpreted as an unsigned integer, is equal to the number of index i between 0 (included) and 32 (excluded) such that the i-th bit of `x` is set. It is very close to the expected functional behavior expressed in English.

Note the `ensures` which directly formalizes the expected functional behavior.

3.1.1 Proving the `count` Function

As given on Figure 2, there is absolutely no chance that an automatic solver can prove the given post-condition. We must add extra annotations in order somehow explain what this algorithm is correct. First let's look at the first operation :

```
x = x - ((x >> 1) & 0x55555555) ;
```

The effect is that after this operation each pair of consecutive bits $b_{2i+1}b_{2i}$, for $0 \leq i < 16$ contains the number of bits originally set among at the same two positions. This can be seen by looking at the four possible cases :

`x`	$=$	$\ldots b_{2i+1}b_i \ldots =$	00	01	10	11
`x>>1 & 0x55..5`	$=$	$\ldots 0b_{2i+1} \ldots =$	00	00	01	01
`x - (x>>1 & 0x55..5)`	$=$		00	01	01	10
		that is	0	1	1	2

This fact can be expressed as the following mathematical formula, where `n` denotes the input bitvector and `x` denotes the value after the assignment (i.e. as in line 5 of Figure 2) :

$$\forall i.0 \leq i < 16 \rightarrow \text{to\_uint}((\text{x} >> 2i) \;\&\; 0\text{x}3) = \text{numof}(\text{nth n})(2i)(2i+2) \tag{1}$$

The other operations aim at computing the sum of each of these pairs of bits, making intermediate sums in parallel, e.g. after the bit-wise operations on lines 7-9 of Figure 2, we can state that

$$\forall i.0 \leq i < 8 \rightarrow \text{to\_uint}((\text{x} >> 4i) \;\&\; 0\text{xF}) = \text{numof}(\text{nth n})(4i)(4i+4) \tag{2}$$

These conjectured formulas can be turned into assertions in the code. However, the proof will not succeed automatically yet, for two reasons:

- Proving each of such assertions need reasoning both on the bit-vectors side and on the arithmetic side, requiring the use of even more intermediate assertions and several provers.
- Proving the assertion after a given step of computation need to make use of the assertion stated after the previous step, with the appropriate choices of indices, e.g. for proving formula (2) above for a given index i, one needs to instantiated formula (1) with indices $2i$ and $2i+1$, a reasoning that remains too clever for automated provers.

In order to guide the provers, we proceed by adding ghost functions, one function for each reasoning step, the function for a given step for index i calling the ghost function for the previous step with the appropriate indices $2i$ and $2i+1$. Each of these ghost functions are annotated with auxiliary assertions to enforce the bridge between the bit-vector side and the arithmetic side.

We give below a verbatim of the annotated code for the bit counting example. An even more complete file is available from the Toccata's gallery[10], where some extra operations (Hamming distance, ascii code) are specified and proved.

[10] `http://toccata.lri.fr/gallery/bitcount.en.html`

bitcount.mlw

```
module BitCounting32

  use import int.Int
  use import int.NumOf
  use import bv.BV32
  use import ref.Ref

  predicate step0 (n x1 : t) =
    x1 = sub n (bw_and (lsr_bv n (of_int 1)) (of_int 0x55555555))

  let ghost proof0 (n x1 : t) (i : int) : unit
    requires { 0 <= i < 16 }
    requires { step0 n x1 }
    ensures { to_uint (bw_and (lsr x1 (2*i)) (of_int 0x03))
              = numof (nth n) (2*i) (2*i + 2) }
  = let i' = of_int i in
    let twoi = mul (of_int 2) i' in
    assert { to_uint twoi = 2 * i };
    assert { to_uint (add twoi (of_int 1)) = to_uint twoi + 1 };
    assert { to_uint (bw_and (lsr_bv x1 twoi) (of_int 0x03))
             = (if nth_bv n twoi then 1 else 0) +
               (if nth_bv n (add twoi (of_int 1)) then 1 else 0)
             = (if nth n (to_uint twoi) then 1 else 0) +
               (if nth n (to_uint twoi + 1) then 1 else 0)
             = numof (nth n) (to_uint twoi) (to_uint twoi + 2) }

  predicate step1 (x1 x2 : t) =
    x2 = add (bw_and x1 (of_int 0x33333333))
             (bw_and (lsr_bv x1 (of_int 2)) (of_int (0x33333333)))

  let ghost proof1 (n x1 x2 : t) (i : int) : unit
    requires { 0 <= i < 8 }
    requires { step0 n x1  }
    requires { step1 x1 x2 }
    ensures  { to_uint (bw_and (lsr x2 (4*i)) (of_int 0x07))
               = numof (nth n) (4*i) (4*i+4) }
  = proof0 n x1 (2*i);
    proof0 n x1 (2*i+1);
    let i' = of_int i in
    assert { ult i' (of_int 8) };
    assert { to_uint (mul (of_int 4) i') = 4*i };
    assert { bw_and (lsr x2 (4*i)) (of_int 0x07)
             = bw_and (lsr_bv x2 (mul (of_int 4) i')) (of_int 0x07)
             = add (bw_and (lsr_bv x1 (mul (of_int 4) i')) (of_int 0x03))
                   (bw_and (lsr_bv x1 (add (mul (of_int 4) i') (of_int 2)))
                           (of_int (0x03)))
             = add (bw_and (lsr x1 (4*i)) (of_int 0x03))
                   (bw_and (lsr x1 ((4*i)+2)) (of_int (0x03))) }

  predicate step2 (x2:t) (x3:t) =
    x3 = bw_and (add x2 (lsr_bv x2 (of_int 4))) (of_int 0x0F0F0F0F)

```

```
53  let ghost proof2 (n x1 x2 x3 : t) (i : int) : unit
54    requires { 0 <= i < 4 }
55    requires { step0 n x1 }
56    requires { step1 x1 x2 }
57    requires { step2 x2 x3 }
58    ensures  { to_uint (bw_and (lsr x3 (8*i)) (of_int 0x0F))
59             = numof (nth n) (8*i) (8*i+8) }
60  = proof1 n x1 x2 (2*i);
61    proof1 n x1 x2 (2*i+1);
62    let i' = of_int i in
63    assert { ult i' (of_int 4) };
64    assert { to_uint (mul (of_int 8) i') = 8*i };
65    assert { to_uint (add (mul (of_int 8) i') (of_int 4)) = 8*i+4 };
66    assert { bw_and (lsr x3 (8*i)) (of_int 0x0F)
67             = bw_and (lsr_bv x3 (mul (of_int 8) i')) (of_int 0x0F)
68             = add (bw_and (lsr_bv x2 (mul (of_int 8) i')) (of_int 0x07))
69                   (bw_and (lsr_bv x2 (add (mul (of_int 8) i') (of_int 4))) (of_int (0x07)))
70             = add (bw_and (lsr x2 (8*i)) (of_int 0x07))
71                   (bw_and (lsr x2 ((8*i)+4)) (of_int (0x07))) }
72
73  predicate step3 (x3:t) (x4:t) =
74    x4 = add x3 (lsr_bv x3 (of_int 8))
75
76  let ghost proof3 (n x1 x2 x3 x4 : t) (i : int) : unit
77    requires { 0 <= i < 2 }
78    requires { step0 n x1 }
79    requires { step1 x1 x2 }
80    requires { step2 x2 x3 }
81    requires { step3 x3 x4 }
82    ensures  { to_uint (bw_and (lsr x4 (16*i)) (of_int 0x1F))
83               = numof (nth n) (16*i) (16*i+16) }
84  = proof2 n x1 x2 x3 (2*i);
85    proof2 n x1 x2 x3 (2*i+1);
86    let i' = of_int i in
87    assert { ult i' (of_int 2) };
88    assert { to_uint (mul (of_int 16) i') = 16*i };
89    assert { to_uint (add (mul (of_int 16) i') (of_int 8)) = 16*i+8 };
90    assert { bw_and (lsr x4 (16*i)) (of_int 0x1F)
91             = bw_and (lsr_bv x4 (mul (of_int 16) i')) (of_int 0x1F)
92             = add (bw_and (lsr_bv x3 (mul (of_int 16) i')) (of_int 0x0F))
93                   (bw_and (lsr_bv x3 (add (mul (of_int 16) i') (of_int 8))) (of_int (0x0F)))
94             = add (bw_and (lsr x3 (16*i)) (of_int 0x0F))
95                   (bw_and (lsr x3 ((16*i)+8)) (of_int (0x0F))) }
96
97  predicate step4 (x4:t) (x5:t) =
98    x5 = add x4 (lsr_bv x4 (of_int 16))
99
100 let ghost prove (n x1 x2 x3 x4 x5 : t) : unit
101   requires { step0 n x1 }
102   requires { step1 x1 x2 }
103   requires { step2 x2 x3 }
104   requires { step3 x3 x4 }
105   requires { step4 x4 x5 }
```

```
106     ensures { to_uint (bw_and x5 (of_int 0x3F)) = numof (nth n) 0 32 }
107   = proof3 n x1 x2 x3 x4 0;
108     proof3 n x1 x2 x3 x4 1;
109     assert { bw_and x5 (of_int 0x3F)
110              = add (bw_and x4 (of_int 0x1F)) (bw_and (lsr_bv x4 (of_int 16)) (of_int 0x1F))
111              = add (bw_and (lsr x4 0) (of_int 0x1F)) (bw_and (lsr x4 16) (of_int 0x1F)) }
112
113   let count (n : t) : t
114     ensures { to_uint result = numof (nth n) 0 32 }
115   = let x = ref n in
116     (* x = x - ( (x >> 1) & 0x55555555) *)
117     x := sub !x (bw_and (lsr_bv !x (of_int 1)) (of_int 0x55555555));
118     let ghost x1 = !x in
119     (* x = (x & 0x33333333) + ((x >> 2) & 0x33333333) *)
120     x := add (bw_and !x (of_int 0x33333333))
121              (bw_and (lsr_bv !x (of_int 2)) (of_int (0x33333333)));
122     let ghost x2 = !x in
123     (* x = (x + (x >> 4)) & 0x0F0F0F0F *)
124     x := bw_and (add !x (lsr_bv !x (of_int 4))) (of_int 0x0F0F0F0F);
125     let ghost x3 = !x in
126     (* x = x + (x >> 8) *)
127     x := add !x (lsr_bv !x (of_int 8));
128     let ghost x4 = !x in
129     (* x = x + (x >> 16) *)
130     x := add !x (lsr_bv !x (of_int 16));
131
132     prove n x1 x2 x3 x4 !x;    (* proceed with the proof *)
133
134     (* return (x & 0x0000003F) *)
135     bw_and !x (of_int 0x0000003F)
136
137 end
```

end of `bitcount.mlw`

3.1.2 Proof Results

The results of all these provers with their variants are given in Figure 3. These results allow us to identify the few VCs which are specifically discharged by the SMTLIB bit-vector support of CVC4 and Z3:

- `proof0.3`, corresponding to the first equality of the assertion on lines 20-22
- `proof1.7.2`: corresponds to equality on lines 43-46
- `proof2.10.2`: corresponds to equality on lines 67-69
- `proof3.12.2`: corresponds to equality on lines 91-93
- `prove.11.1`: corresponds to equality on lines 109-110

As expected, they correspond to auxiliary assertions expressing properties at the bit-wise level. The other assertions, pre- and post-conditions, are discharged by other SMT solvers, including the "noBV" variants of CVC4 and Z3.

Proof obligations		Alt-Ergo (0.99.1)	CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.0)	Z3 (4.4.0 noBV)
VC for proof0	1. assertion	0.06	(5s)	(5s)	(5s)	(5s)
	2. assertion	0.04	(5s)	0.06	(5s)	(5s)
	3. assertion	(5s)	0.14	(5s)	(5s)	(1000M)
		0.04	(5s)	0.09	(5s)	(5s)
		0.80	(5s)	0.08	0.02	(5s)
	4. postcondition	0.44	(5s)	0.06	(5s)	(5s)
VC for proof1	1. precondition	0.03	0.03	0.03	0.01	0.01
	2. precondition	0.03	0.03	0.04	0.01	0.01
	3. precondition	0.03	0.03	0.04	0.01	(5s)
	4. precondition	0.03	0.03	0.04	0.02	(5s)
	5. assertion	0.06	(5s)	0.06	(5s)	(5s)
	6. assertion	0.97	0.08	(5s)	(5s)	(5s)
	7. assertion	0.04	(5s)	0.06	(5s)	(5s)
		(5s)	0.03	(5s)	0.02	(5s)
		0.05	(5s)	1.23	(5s)	(5s)
	8. postcondition	(5s)	0.08	0.08	(5s)	(5s)
VC for proof2	1. precondition	0.04	0.05	0.04	0.01	0.02
	2. precondition	0.03	0.05	0.04	0.02	0.02
	3. precondition	0.03	0.05	0.02	0.02	0.02
	4. precondition	0.03	0.04	0.04	0.02	(5s)
	5. precondition	0.02	0.05	0.03	0.02	(5s)
	6. precondition	0.02	0.04	0.03	0.01	(5s)
	7. assertion	0.06	(5s)	0.04	(5s)	(5s)
	8. assertion	0.63	0.08	(5s)	(5s)	(5s)
	9. assertion	0.75	0.07	0.11	(5s)	(5s)
	10. assertion	0.06	(5s)	0.06	(5s)	(5s)
		(5s)	0.07	(5s)	0.02	(5s)
		0.04	(5s)	0.06	(5s)	(5s)
	11. postcondition	(5s)	0.09	0.08	(5s)	(5s)
VC for proof3	1. precondition	0.05	0.05	0.04	0.02	0.02
	2. precondition	0.05	0.06	0.04	0.02	0.02
	3. precondition	0.05	0.05	0.04	0.02	0.02
	4. precondition	0.05	0.06	0.04	0.02	0.01
	5. precondition	0.05	0.05	0.04	0.02	(5s)
	6. precondition	0.05	0.05	0.04	0.02	(5s)
	7. precondition	0.05	0.05	0.04	0.02	(5s)
	8. precondition	0.05	0.04	0.04	0.01	(5s)
	9. assertion	0.05	(5s)	0.07	(5s)	(5s)
	10. assertion	0.98	0.08	(5s)	(5s)	(5s)
	11. assertion	0.05	0.11	0.07	(5s)	(5s)
	12. assertion	0.07	(5s)	0.04	(5s)	(5s)
		(5s)	0.05	(5s)	0.02	(5s)
		0.06	(5s)	0.06	(5s)	(5s)
	13. postcondition	(5s)	0.09	0.08	(5s)	(5s)
VC for prove	1. precondition	0.03	0.04	0.03	0.01	0.01
	2. precondition	0.03	0.05	0.05	0.02	0.02
	3. precondition	0.05	0.04	0.04	0.02	0.02
	4. precondition	0.04	0.05	0.04	0.02	0.01
	5. precondition	0.05	0.04	0.05	0.02	0.02
	6. precondition	0.04	0.02	0.02	0.00	(5s)
	7. precondition	0.04	0.03	0.04	0.01	(5s)
	8. precondition	0.04	0.03	0.04	0.02	(5s)
	9. precondition	0.04	0.03	0.04	0.01	(5s)
	10. precondition	0.04	0.03	0.03	0.01	(5s)
	11. assertion	(5s)	0.04	(5s)	0.02	(5s)
		0.04	(5s)	0.05	(5s)	(5s)
	12. postcondition	(5s)	0.27	0.07	(5s)	(5s)
VC for count	1. precondition	0.04	0.04	0.06	0.02	0.18
	2. precondition	0.06	0.04	0.05	0.05	0.75
	3. precondition	0.05	0.05	0.06	0.05	0.49
	4. precondition	0.05	0.06	0.05	0.05	0.81
	5. precondition	0.04	0.04	0.05	0.09	0.46
	6. postcondition	0.05	0.04	0.04	0.01	(1000M)

Figure 3: Proofs for Counting Bits example

```
theory S

  use export set.Fsetint  (* from why3's standard library *)

  function succ (set int) : set int
  axiom succ_def:
    forall s: set int, i: int. mem i (succ s) ↔ i >= 1 ∧ mem (i-1) s

  function pred (set int) : set int
  axiom pred_def:
    forall s: set int, i: int. mem i (pred s) ↔ i >= 0 ∧ mem (i+1) s

end
```

Figure 4: Theory of finite sets of integers with succ and pred operations

3.2 The n-Queens Problem

We show how we can handle the tricky code for solving the n-queens problem, that we presented in introduction.

We follow the guidelines of Filliâtre's version [14], where an abstract version of the code is proved, using automated provers but also the Coq proof assistant for a few VCs (see also `http://toccata.lri.fr/gallery/queens.en.html`). The verified code is specified and coded in Why3, where the C type `int` is replaced by the Why3 type `set int` of sets of integers. This is to reflect the intended interpretation of a C machine integer x: the set of indices i such that the i-th bit of x is set. Each bit-wise operations is thus interpreted into a set operation:

- `0` represents the empty set
- `~(~0<<n)` denotes the set $\{0, \ldots, n-1\}$
- the "rightmost bit trick" `x&-x` denotes the smallest element of `x` (assuming `x` is non empty)
- `x & ~y` denotes the set difference of `x` and `y`
- `x + y` denotes the set union, assuming that `x` and `y` are disjoint
- `x*2` (resp. `x/2`) the denotes the set obtained by incrementing (reps. decrementing) by 1 each element of `x`

We propose here a refinement of Filliâtre's Why3 code for these operations. We start by introducing a small theory of finite sets of integers, obtained from the one in Why3's standard library, augmented with operations `succ` and `pred` to increment (resp. decrement) each element in a set. This is shown on Figure 4. We then provide, as shown on Figures 6 a module defining a new type `t` made of a 32-bit vectors with an attached ghost model, a set of integers. We use the type invariant mechanism of Why3 to relate those bit-vectors to this ghost view as a set of integers.

Notice that the code for `rightmost_bit_trick` slightly differs from our version of Section 2.3. Since we have the model as a set, we don't need a ghost function anymore to compute the minimum of a set. The proofs proceeds as expected. The results are shown on Figure 5. The only functions that needs a precise bit-vector reasoning are `rightmost_bit_trick` and `below`.

Proof obligations		**Alt-Ergo (0.99.1)**	**CVC4 (1.4)**	**CVC4 (1.4 noBV)**	**Z3 (4.4.0)**	**Z3 (4.4.0 noBV)**
VC for empty		0.03	(5s)	0.04	(5s)	0.69
VC for is_empty		0.08	(5s)	0.07	(5s)	(5s)
VC for remove_singleton		1.58	(5s)	0.17	(5s)	(5s)
VC for add_singleton		(5s)	(5s)	0.07	(5s)	(5s)
VC for mul2		2.07	(5s)	0.43	(5s)	(5s)
VC for div2		0.26	(5s)	0.20	(5s)	(5s)
VC for diff		0.40	(5s)	0.05	(5s)	(5s)
VC for rightmost_bit_trick	1. assertion	(5s)	(5s)	0.07	(5s)	(5s)
	2. assertion	0.12	(5s)	0.05	(5s)	0.88
	3. assertion	(5s)	0.02	(5s)	0.02	(5s)
	4. assertion	(5s)	0.03	(5s)	0.06	(5s)
	5. type invariant	(5s)	(5s)	0.20	(5s)	(5s)
	6. postcondition	0.04	0.01	0.03	0.01	0.00
VC for below	1. assertion	(5s)	0.02	0.07	0.01	(1000M)
	2. type invariant	(5s)	(5s)	0.05	(5s)	(5s)

Figure 5: Proof results for BitsAsSets

The rest of the code, that uses the module `BitsAsSets`, is similar to Filliâtre. The full code for this case study, with all the required annotations to performs the proofs automatically, is available from the Toccata's gallery[11], together with the detailed proof results.

[11] `http://toccata.lri.fr/gallery/queens_bv.en.html`

```
module BitsAsSets

  use import S
  use import bv.BV32

  type t = { bv : BV32.t; ghost mdl: set int; }
  invariant { forall i: int. (0 <= i < size ∧ nth self.bv i) ↔ mem i self.mdl }

  let empty () : t ensures { is_empty result.mdl }
  = { bv = zero; mdl = empty }

  let is_empty (x:t) : bool ensures { result  ↔ is_empty x.mdl }
  = assert {is_empty x.mdl → BV32.eq x.bv zero}; x.bv = zero

  let remove_singleton (a b: t) : t
    requires { b.mdl = singleton (min_elt b.mdl) }
    requires { mem (min_elt b.mdl) a.mdl }
    ensures  { result.mdl = remove (min_elt b.mdl) a.mdl }
  = { bv = bw_and a.bv (bw_not b.bv); mdl = remove (min_elt b.mdl) a.mdl }

  let add_singleton (a b: t) : t
    requires { b.mdl = singleton (min_elt b.mdl) }
    ensures  { result.mdl = S.add (min_elt b.mdl) a.mdl }
  = { bv = bw_or a.bv b.bv; mdl = S.add (min_elt b.mdl) a.mdl }

  let mul2 (a: t) : t ensures { result.mdl = remove size (succ a.mdl) }
  = { bv = lsl_bv a.bv (of_int 1); mdl = remove size (succ a.mdl) }

  let div2 (a: t) : t ensures { result.mdl = pred a.mdl }
  = { bv = lsr_bv a.bv (of_int 1); mdl = pred a.mdl }

  let diff (a b: t) : t ensures { result.mdl = diff a.mdl b.mdl }
  = { bv = bw_and a.bv (bw_not b.bv); mdl = diff a.mdl b.mdl }

  let rightmost_bit_trick (a: t) : t
    requires { not (is_empty a.mdl) }
    ensures  { result.mdl = singleton (min_elt a.mdl) }
  = let ghost n = min_elt a.mdl in
    let ghost n_bv = of_int n in
    assert { eq_sub_bv a.bv zero zero n_bv };
    assert { nth_bv a.bv n_bv };
    assert { nth_bv (neg a.bv) n_bv };
    let res = bw_and a.bv (neg a.bv) in
    assert { eq_sub_bv res zero (add n_bv (of_int 1)) (sub (of_int 31) n_bv ) };
    { bv = res; mdl = singleton n }

  let below (n: BV32.t) : t
    requires { BV32.ule n (BV32.of_int 32) }
    ensures  { result.mdl = interval 0 (to_uint n) }
  = let res = bw_not (lsl_bv ones n) in
    assert { forall i. nth_bv res i ↔ ult i n };
    { bv = res;
      mdl = interval 0 (to_uint n) }

end
```

Figure 6: 32-bit vectors, with a ghost model as a finite set of integers

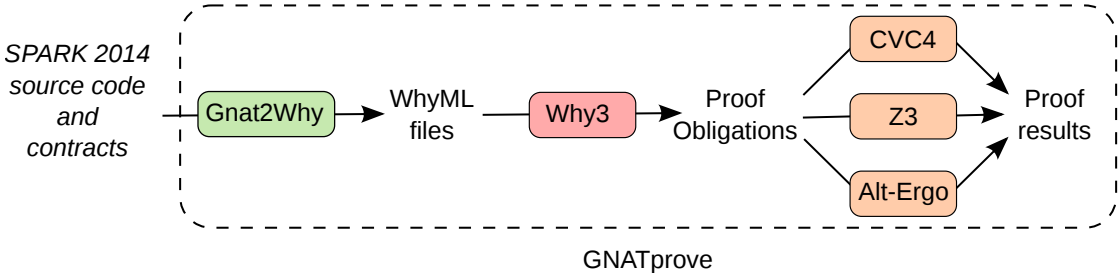


Figure 7: Deductive verification in SPARK 2014

4 The "Bitwalker" case study, using SPARK2014

4.1 Adding Support for Bit-Vectors in SPARK2014

Ada is a programming language targeted at real-time embedded software which requires a high level of safety, security, and reliability. In particular, it provides a wide range of run-time checks, for example for buffer overflows, and has a verbose syntax that makes it easy to read and debug. For these reasons, Ada is nowadays used in domains where software cannot be allowed to fail.

Ada 2012 is the latest version of the Ada language [1]. It contains new features for specifying the behavior of programs, such as subprogram contracts and type invariants. When given a specific compilation switch, the Ada compiler can turn these constructs into assertions to check at run time. Thanks to this switch, the conformance of the implementation of a program to its specification can be checked dynamically during the process of unit testing.

SPARK, co-developed by Altran and AdaCore, is a subset of Ada targeted at formal verification [10]. Its restrictions ensure that the behavior of a SPARK program is unambiguously defined (unlike Ada). It excludes constructions that cannot easily be verified by automatic tools. 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.

SPARK 2014, co-developed by Altran and AdaCore, is a subset of Ada 2012 targeted at formal verification [21]. It comprises most of the Ada 2012 language excluding constructs which are not easily amenable to sound static verification. Features such as pointers, side effects in expressions, aliasing, goto statements, controlled types (e.g. types with finalization) and exception handling are excluded.

SPARK 2014 is designed so that both the flow analysis – checking that there is no access to uninitialized variables and that global variables and subprogram parameters are accessed appropriately – and the proof of program – checking the absence of run-time errors and the conformance to the contract – can be checked. It provides dedicated features that are not part of Ada 2012. In particular, contracts can also contain information about data dependencies, information flows, state abstraction, and data and behavior refinement that can be checked by the GNATprove tool. Essential constructs for formal verification such as loop variants and invariants have also been introduced.

As described in Figure 7, to formally prove a SPARK 2014 program, GNATprove uses WhyML as an intermediate language. The SPARK program is translated into an equivalent WhyML program which can then be verified using the Why3 tool.

4.1.1 Integer Types in Ada

Ada's very rich type system allows to define various kinds of integer types. They are mostly two variants, namely *signed* and *modular* integer types.

Signed types Signed integer types are usually defined using a range, e.g.:

```
type Int_10 is range 1 .. 10;
```

Here, `Int_10` represents the set of integers between 1 and 10. Signed integer types have a *base type* which is another signed integer type. Typically, it is a type corresponding to the smallest possible machine words in which all the value of the range (in the classical 2-complement representation) fit. The choice of the base type is compiler-dependent. For example, the base type of `Int_10` will most probably be the type of all signed 8-bits words, ranging from -2^7 to $2^7 - 1$. The semantics of Ada specifies that each single operation on a signed type will raise an exception whenever the result overflows its base type. Additionally, when the result of such a computation is stored back into a variable or given as input to a function call, it is checked that the type's bounds, here 1 and 10, are not exceeded. Example

```
A : Int_10 := 10;
B : Int_10 := (A + 1) - 1;      --  No overflow here
C : Int_10 := (A + 127) - 127;  --  Exception: overflow in the first addition
D : Int_10 := A + 1;            --  Exception: type range exceeded
```

Modular types Modular integer types are defined by specifying a modulo. For example

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

defines a type `BV8` that contains integers between 0 and $2^8 - 1$. No overflow will ever occur when computing with it. Indeed, computations on modular types use, as their name suggests, a modular semantics which brings the resulting value back between 0 and the modulus minus one. For example

```
A : BV8 := 128;
B : BV8 := A + 128;  -- No overflow, result is 0
```

Additionally, one can specify a range on a sub-type of a modular type. For example,

```
type BV8_10 is new BV8 range 1 .. 10;
```

defines a type `BV8_10` that, like `Int_10`, contains only integers between 1 and 10. Like signed types, the range is checked at assignment and parameter passing:

```
A : BV8_10 := 10;
B : BV8_10 := (A + 1) - 1;  --  No overflow here
C : BV8_10 := A + 255;      --  No overflow either, result is 9
D : BV8_10 := A + 1;        --  Exception: type range is exceeded
```

The package `Interfaces` from Ada's standard library proposed 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 operations Bit-wise operators are supported in Ada, but only on modular integer types. Bit-wise boolean operations are written as infix operators `and`, `or`, `xor`, `not`. These operations operate as expected when the modulus is a power of two (see Barnes [1] for further information). This is always the case in the remaining of this report.

Ada also provides, in its standard library, functions for shifts but also for *rotations*: `Shift_Left`, `Shift_Right`, `Shift_Right_Arithmetic`, `Rotate_Left`, `Rotate_Right`. These operations 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 a modular but of type `Natural`, that is the signed integer type of only non-negative values defined in Ada's standard library as

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

Note that when the second argument is larger that the number of bits, shift operations behave like in SMTLIB, whereas for rotations the result is as if several turns were done. For example:

```
A : BV8 := Shift_Left(1,2);  -- value is 4
B : BV8 := Shift_Left(1,8);  -- value is 0
C : BV8 := Shift_Left(1,9);  -- value is 0
D : BV8 := Rotate_Left(3,2); -- value is 12
E : BV8 := Rotate_Left(3,7); -- value is 129
F : BV8 := Rotate_Left(3,9); -- value is 6
```

4.1.2 Handling of Ada's Integer Types in SPARK 2014

GNATprove translates each Ada variable, resp. each expression, into a Why3 variable, resp. expression, of some adequate type. We focus here on the integer types, for other things like translation of arrays, of records, of procedures, see [17].

Signed types Variables (resp. expressions) of signed integer types are translated into variables (resp. expressions) of the Why3 type `int` of unbounded mathematical integers. Therefore their range is not part of their Why3 type and has to be enforced in some other way. For each Ada sub-expression e whose base type is b, we need to check absence of overflow during computation of the top operator in e. We proceed by translating e into a Why3 expression of the form

```
let temp = <translation of e> in
assert { in_range_b temp };
... temp ...
```

where

```
predicate in_range_b (n:int) = i <= n <= j
```

and i and j are the bounds for the base type b. Each basic operator like addition, subtraction is thus directly translated into the corresponding operation on `int`. This takes into account only base types, so to also check the range in the case of an assignment $X := e$ for a variable X of type t, another assertion is inserted in the code has follows.

```
let temp = <translation of e> in
assert { in_range_t temp };
x := temp;
```

where `in_range_t` checks against the bounds given in the declaration of t.

Example The Ada code of Example 4.1.1 is translated into Why3 as

```
1  predicate in_range_short_short_integer (n:int) = -128 <= n <= 127
2  predicate in_range_int10 (n:int) = 1 <= n <= 10
3  let a : ref int = 10 in
4  let temp1 = !a + 1 in
5  assert { in_range_short_short_integer temp1 };
6  let temp2 = temp1 - 1 in
7  assert { in_range_short_short_integer temp2 };
8  assert { in_range_int10 temp2 };
9  let b : ref int = temp2 in
10 let temp3 = !a + 127 in
11 assert { in_range_short_short_integer temp3 };
12 let temp4 = temp3 - 127 in
13 assert { in_range_short_short_integer temp4 };
14 assert { in_range_int10 temp4 };
15 let c : ref int = temp4 in
16 let temp5 = !a + 1 in
17 assert { in_range_short_short_integer temp5 };
18 assert { in_range_int10 temp5 };
19 let d : ref int = temp5 + 1 in
20 ...
```

The assert at line 11 will generate the failing overflow check, and the assert at line 18 will generate to failing range check.

Note that consequently, to prove the absence of overflow and validity of ranges for signed types, we rely on the theory of integer arithmetic provided by the SMT solvers.

Modular types 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 bitwise operations is directly the equivalent introduced in our Why3 theory in Section 2.2.2. The translation of shifts and rotations is just slightly more complex because their second argument in Ada is a signed type and not a modular. We thus translate

```
... Shift_Left(X,Y) ...
```

as

```
let temp = lsl_bv X (if Y < size then (of_int Y) else size_bv) in
...
```

The guard `Y < size` is needed because the `of_int` operator converts an integer into a bit-vector modulo 2^{size}, hence the translation would be wrong without the guard for large values of Y.

Example The Ada code

```
A : BV8 := 42;
B : BV8 := A + 1;
C : BV8 := A or B;
D : BV8 := Shift_Left(C,2);
```

is translated into

```
let a : ref BV8.t = BV8.of_int 42 in
let b : ref BV8.t = BV8.add !a (BV8.of_int 1) in
let c : ref BV8.t = BV8.bw_or !a !b in
let d : ref BV8.t = BV8.lsl_bv !c (BV8.of_int 2) in
...
```

In particular there are no run-time checks at all on such a code.

4.1.3 Translation of conversions

In SPARK 2014 one can convert between any two numeric types as long as the value that is converted is in the range of the target type (using rounding in the case of float to integer conversions). GNATprove will translate any such conversion in SPARK 2014 into a conversion in Why3 between the corresponding representation types (`int`, `BV8`...). A range check will be inserted to ensure that the conversion yield a valid value of the target type.

In the case of a conversion between a modular type and non modular type GNATprove will insert an intermediate conversion to int[12]. Indeed, in the Why3 theory of bit-vectors, as well as the SMT-LIB theory, there are only conversions to and from integers. For a conversion between two modular types represented by two different bit-vector types, the appropriate theory of bit-vector conversions (see 2.2.11) is used. GNATprove will insert a range check occurring within the biggest type involved in the conversion (int in the case of modular vs non-modular).

It is possible to do unchecked conversions in Ada but, as their name suggest, for those SPARK 2014 doesn't guarantee the validity of the resulting value (13.9 in SPARK 2014 Reference Manual).

4.1.4 External Axiomatization for High-level Specification in SPARK

To attach high-level formal specifications to Ada sub-programs, in a similar way as we did in Why3, we also provide a so-called *external axiomatization* for additional bit-vector operations useful for such specifications (see SPARK 2014 User's Guide [13], §8.5). This amounts to declare Ada functions in some Ada package specification, and declare a mapping of these functions to some Why3 logic functions or predicates. Here the package specification we declare for those additional bit-vector functions that we will use in the SPARK 2014 version of Bitwalker.

```
package Bvext
with SPARK_Mode, Ghost
is
   pragma Annotate (GNATProve, External_Axiomatization);
   function Nth (X : Unsigned_64; Pos : Natural) return Boolean with Import;
   function Nth (X : Unsigned_8; Pos : Natural) return Boolean with Import;
   function Nth_Bv (X, Pos : Unsigned_64) return Boolean with Import;
   function Nth_Bv (X, Pos : Unsigned_8) return Boolean with Import;
   function Eq_Sub (X, Y : Unsigned_64; I, N : Natural) return Boolean with Import;
   function Eq_Sub_Bv (X, Y, I, N : Unsigned_64) return Boolean with Import;
   function Eq (X, Y : Unsigned_64) return Boolean with Import;
end Bvext;
```

The `pragma` specify that the package uses an external axiomatization, GNATprove will then look for the definitions of the functions in a separate WhyML file. In the WhyML file, the two first functions `Nth` are

[12] With the future use of native support of floating point types by SMT-Solvers in GNATprove, the conversions between modulars and floats will be direct.

[13] `http://docs.adacore.com/spark2014-docs/html/ug/index.html`

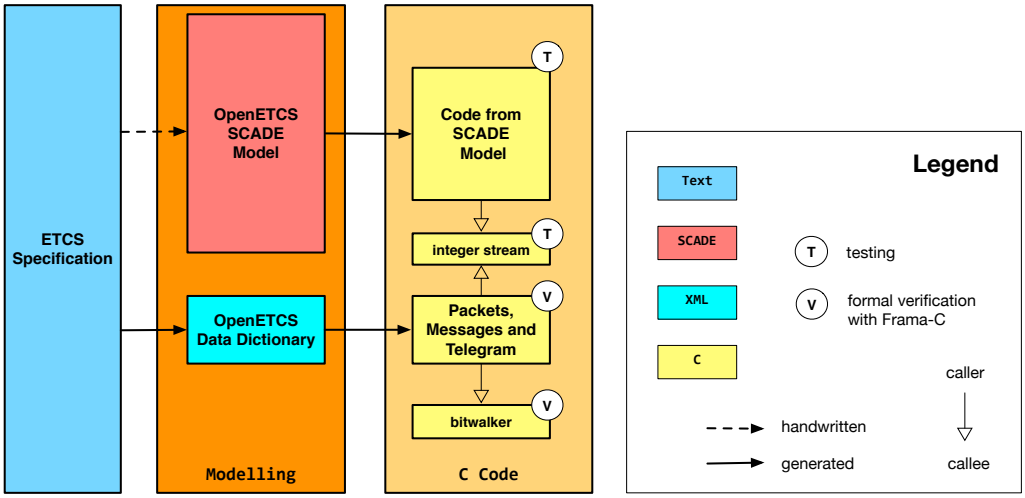


Figure 8: BitWalker and Frama-C in the OpenETCS project

mapped to Why3 functions `BV64.nth` and `BV8.nth` (from Section 2.2.1), and the next two are mapped to `BV64.nth_bv` and `BV8.nth_bv` (from Section 2.2.8). `Eq_Sub` is mapped to `BV64.eq_sub`, `Eq_Sub_Bv` to `BV64.eq_sub_bv` and `Eq` to `BV64.eq` (from Section 2.2.9). In other words, this external axiomatization allows the user to attach Ada contracts to bit-vector sub-programs, similar to what could have been done directly in Why3.

In order to ensure that these functions are only used for specification and proof, each functions is marked as imported (i.e., specify that there is no Ada body for them) and the package is ghost.

Note that such an external axiomatization is a candidate to be distributed with SPARK, to be used by regular users.

4.2 BitWalker: peeking and poking bits from/to a stream

The original C version of the BitWalker was provided by Siemens in the context of the ITEA 2 project OpenETCS[14]. The version presented in this report was rewritten by Fraunhofer FOKUS[15] to simplify the formal verification with Frama-C/WP [11]. Figure 8 shows the place of the BitWalker and Frama-C in the "grand scheme" of OpenETCS.

The formal specification of the BitWalker in Frama-C's specification language ACSL [4] and results of the formal verification are part of the upcoming OpenETCS report *D4.3.2 Final Report on Validation and Verification Report of Implementation/Code*[16]. The formal specification relies on a theory of bitvectors designed in the proof assistant Coq, and a significant part of the proofs are done interactively within Coq.

Figure 9 presents the excerpt of the C source code for copying a 64-bit value from the byte stream to a 64-bit unsigned integer. The main function is `Bitwalker_Peek`. The expected behavior can be expressed at a high-level by saying that the integer value of the result is the value read in the byte stream `addr` 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. This informal behavior is illustrated on Figure 10.

[14] `http://openetcs.org`
[15] `http://www.fokus.fraunhofer.de`
[16] `https://github.com/openETCS/governance/wiki/State-of-Deliverables`

```
// returns the bit at index [left] in [byte]
static inline int PeekBit8(uint8_t byte, uint32_t left) {
  uint8_t mask = ((uint8_t) 1) << (7u - left);
  uint8_t flag = byte & mask;
  return flag != 0;
}

// returns the bit at index [left] in the byte sequence [addr]
int PeekBit8Array(uint8_t*  addr, uint32_t size, uint32_t left) {
  return PeekBit8(addr[left / 8], left % 8);
}

// 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 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;
}
```

Figure 9: The BitWalker, C version, the Peek function

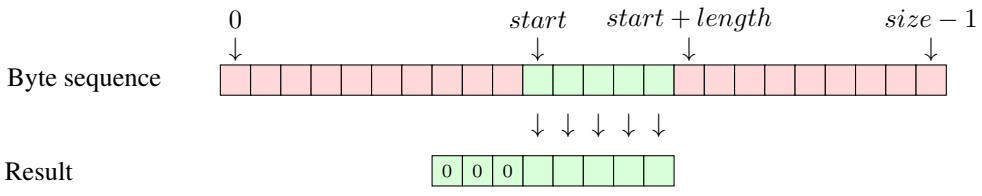


Figure 10: Schematic view of the Peek function (on 8-bit instead of 64-bit)

```
1  type Byte_Sequence is array (Natural range <>) of Unsigned_8;
2
3  function Nth8_Stream (Stream : Byte_Sequence; Pos : Natural) return Boolean is
4    (Nth (Stream (Pos / 8), 7 - (Pos rem 8)))
5  with Pre => Stream'First = 0 and then (Pos / 8 <= Stream'Last), Ghost;
6
7  function Peek  (Start, Length : Natural; Addr : Byte_Sequence) return Unsigned_64
8  with
9    Pre => Addr'First = 0 and then
10          Length <= 64 and then
11          Start + Length <= Natural'Last and then
12          8 * Addr'Length <= Natural'Last,
13   Contract_Cases =>  (
14          Start + Length > 8 * Addr'Length => Peek'Result = 0,
15          Start + Length <= 8 * Addr'Length =>
16            (for all I in 0 .. Length - 1 =>
17                Nth8_Stream (Addr, Start + Length - I - 1) = Nth (Peek'Result, I))
18            and then
19            (for all I in Length .. 63 => not Nth (Peek'Result, I)));
```

Figure 11: Ada specification of `Peek` function

The code of `Bitwalker_Peek` does not make use of low-level bitwise operators, but instead calls auxiliary functions. On the contrary, the codes of low-level auxiliary functions `PeekBit8` and `PokeBit64` make use of bitwise operators, so there is a need at some point to related those bitwise operations with more high-level arithmetic notions.

In the following, we propose a SPARK code equivalent to the C code of Figure 9, with appropriate specifications.

4.2.1 Specification of Bitwalker `Peek`

Let's start with the specification of the main function `Peek`. It is given in Figure 11. A first difference between the C and Spark version appears in the types: in C all the parameters are unsigned types. In Ada, it would not be idiomatic to use unsigned types for the variables that are used as indexes in the array `Addr`. This is why in our SPARK version of the code, on line 7, `Start` and `Length` are signed, whereas the contents of the array `Addr` are 8-bit modular types, and the result of `Peek` is a 64-bit modular. Note also that the parameter `size` of the C code is not present, because it corresponds to `Addr'Length` in Ada.

The pre-condition starts on line 9, by specifying that the first index of our byte sequence is 0, as in the C code. On line 10 we bound `Length`, the number of bits to copy, by 64. The pre-condition on line 11 requires that the last bit to copy is in the bounds of the byte sequence, and finally on line 12, to avoid any arithmetic overflow, we add a requirement that eight times the length of the byte sequence, indeed the number of bits, is not larger than the range of signed integers.

The post-condition starts on line 13. It is made of two disjoint contract cases, depending on whether the last bit to copy is in the bounds of the byte sequence or not. In the first case, where it exceeds the length of the byte sequence, we specify that the default value 0 is returned. In the other case, we specify two things: first, on lines 16-17, we specify the i-th bit of the result, for $0 \leq i < Length$ is equal to the bit of the sequence at position `Start` + `Length` $- i - 1$, as shown on Figure 10. The n-th bit of a `ByteSequence` is specified by the auxiliary function `Nth8_Stream` given on line 3 of Figure 11, whereas

```
1  function Peek (Start, Length : Natural; Addr : Byte_Sequence) return Unsigned_64 is
2  begin
3    if Start + Length > 8 * Addr'Length then
4      return 0;
5    end if;
6    declare
7      Retval : Unsigned_64 := 0;
8      Flag   : Boolean;
9    begin
10     for I in 0 .. Length - 1 loop
11       pragma Loop_Invariant
12         (for all J in Length - I .. Length - 1 =>
13            Nth8_Stream (Addr, Start + Length - J - 1) = Nth (Retval, J));
14       pragma Loop_Invariant
15         (for all J in Length .. 63 => not Nth (Retval, J));
16       Flag := PeekBit8Array (Addr, Start + I);
17       Retval := PokeBit64 (Retval, (64 - Length) + I, Flag);
18     end loop;
19     return Retval;
20   end;
21 end Peek;
```

Figure 12: Ada code for function `Peek`

the `Nth` function is the one defined on our external axiomatization. Finally, on line 19, we specify that the other bits of the result are set to zero.

The Ada code of `Peek`, translated from the C code of Figure 9, is given on Figure 12. On lines 11-15 are the loop invariants that we inserted in order to prove that the code satisfies the specification. These are natural generalizations of the post-conditions: the invariant on lines 12-13 specifies the bits that are already copied, the invariant on lines 14-15 specifies the bits of index larger or equal to Length remain zero all the time.

The auxiliary functions `PeekBit8Array` and `PokeBit64`, together with an additional auxiliary function `PeekBit8` are specified on Figure 13. With the function `Nth` we can specify PeekBit8 in the following way : for a value of `Left` in $\{0..7\}$ the result is the result of `Nth` on `Byte` at $7-$`Left`. The next function, `PeekBit8Array`, is quite similar in body and specification to PeekBit8. It takes a sequence of bytes and a value `left` smaller than the number of bits in the stream. It returns the value (here again as a boolean) of the bit at position `Left` in the stream. The next function, `PokeBit64`, writes a bit in an `Unsigned_64` value at the given position `Left`. In order to specify this we need to: first write that the mentioned bit is correctly set after the function is called, and then, not to forget that for all other indices nothing changed.

The specifications of these auxiliary functions and the given loop invariants allow us to automatically prove the body of `Peek`, provided that we set the time limit for the provers to one minute.

The results of the provers on `Peek` are displayed in Figure 14. Part of the proof obligations are run-time checks automatically inserted by GNATprove. For example the first two proof obligations verify that the contract cases are disjoint and complete, respectively, and the third one checks that `8 * length addr` is in range. We only mention the other six that are not proved by all the provers:

- Obligation 18 checks that the precondition of `Nth8_Stream` holds when called in the loop invariant line 13.

```
function PeekBit8 (Byte : Unsigned_8; Left : Natural) return Boolean
with
  Pre  => Left < 8,
  Post => PeekBit8'Result = Nth (Byte, 7 - Left);

function PeekBit8Array (Addr : Byte_Sequence; Left : Natural) return Boolean
with
  Pre => Addr'First = 0 and then Left < 8 * Addr'Length,
  Post => PeekBit8Array'Result = Nth8_Stream (Addr, Left);

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));
```

Figure 13: Specifications of auxiliary functions for `Peek`

- Obligation 27 checks that the first loop invariant is preserved during the loop.
- Obligation 31 checks that `Start + Length - I - 1` is in range at line 17 in `Peek` specification.
- Obligation 32 checks again that the precondition of `Nth8_Stream` holds but this time when called in the specification of `Peek` line 17.
- Obligation 37 corresponds to the second case of the contract case when `Start + Length` is greater than `8 * Addr'Length`.
- Obligation 46 corresponds to the second case of the contract case when if's condition, line 3 of the body, failed.

There is no proof obligation that are only proved by CVC4 and Z3, this can be easily explained by the fact that the code of the `Peek` function does not make use of bitwise operators directly, but calls auxiliary functions instead.

	Proof obligations	CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.0)	Z3 (4.4.0 noBV)	altergo (0.99.1)
VC for def	1. check	0.04	0.11	0.02	(5s)	0.09
	2. check	0.04	0.11	0.03	(5s)	0.08
	3. precondition	0.04	0.14	0.03	(5s)	0.45
	4. precondition	0.04	0.13	0.02	0.04	0.08
	5. assertion	0.03	0.11	0.01	0.00	0.09
	6. VC for def	0.04	0.07	0.01	0.00	0.07
	7. precondition	0.03	0.07	0.01	0.01	0.09
	8. precondition	0.04	0.07	0.00	0.01	3.18
	9. precondition	0.04	0.08	0.01	0.01	0.10
	10. VC for def	0.04	0.07	0.01	0.01	0.08
	11. precondition	0.04	0.08	0.00	0.01	0.08
	12. VC for def	0.04	0.08	0.00	0.01	0.08
	13. assertion	0.06	0.08	0.00	0.01	0.41
	14. loop invariant init	0.08	0.10	0.02	0.03	0.08
	15. loop invariant init	0.07	0.13	0.01	0.21	0.06
	16. precondition	0.05	0.09	0.02	0.21	0.05
	17. VC for def	0.04	0.04	0.00	0.01	0.04
	18. precondition	0.06	0.11	0.02	0.19	0.06
	19. precondition	0.06	0.09	0.02	(5s)	(5s)
	20. precondition	0.08	0.08	0.01	0.20	0.05
	21. VC for def	0.06	0.04	0.00	0.01	0.04
	22. precondition	0.12	0.07	0.02	0.21	0.05
	23. precondition	0.12	0.06	0.02	0.04	0.04
	24. precondition	0.11	0.08	0.02	0.14	0.06
	25. precondition	0.08	0.07	0.02	0.03	0.07
	26. precondition	0.07	0.06	0.01	0.02	0.05
	27. loop invariant preservation	(5s)	(5s)	0.05	(5s)	1.56
	28. loop invariant preservation	0.95	0.09	0.02	3.83	4.64
	29. assertion	0.06	0.04	0.00	0.01	0.81
	30. VC for def	0.06	0.03	0.00	0.01	0.10
	31. precondition	0.10	0.08	0.01	(5s)	0.15
	32. precondition	0.23	0.09	0.02	(5s)	(5s)
	33. precondition	0.10	0.08	0.02	0.22	0.15
	34. VC for def	0.06	0.04	0.00	0.01	0.12
	35. precondition	0.09	0.08	0.02	0.23	0.16
	36. VC for def	0.06	0.04	0.00	0.01	0.13
	37. assertion	(5s)	0.30	0.04	(5s)	(5s)
	38. assertion	0.06	0.07	0.00	0.01	0.10
	39. VC for def	0.06	0.06	0.00	0.01	0.08
	40. precondition	0.08	0.09	0.01	0.03	0.08
	41. precondition	0.08	0.10	0.01	0.04	0.04
	42. precondition	0.08	0.12	0.01	0.02	0.07
	43. VC for def	0.06	0.07	0.00	0.00	0.08
	44. precondition	0.09	0.11	0.01	0.20	0.10
	45. VC for def	0.06	0.03	0.00	0.01	0.05
	46. assertion	0.09	0.07	0.01	(5s)	0.06

Figure 14: Provers results on Peek

```
function PeekBit8 (Byte : Unsigned_8; Left : Natural) return Boolean is
   Mask : constant Unsigned_8 := Shift_Left (1, 7 - Left);
   Flag : constant Unsigned_8 := Byte and Mask;
begin
   return Flag /= 0;
end PeekBit8;

function PeekBit8Array (Addr : Byte_Sequence; Left : Natural) return Boolean is
begin
  return PeekBit8 (Addr (Left / 8), Left rem 8);
end PeekBit8Array;

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;
```

Figure 15: Bodies of auxiliary functions for `Peek`

4.2.2 Verification of the auxiliary functions

The bodies of the three auxiliary functions are given on Figure 15. The body of `PeekBit8` is a literal translation of the C code into Spark. No additional assertion is needed for the tool to verify its specification. Similarly, GNATprove verifies `PeekBit8Array` specification without any additional assertion. The verification of `PokeBit64` is more involved, we need to help the provers with some assertions. We are now in front of the typical problem of mixing modulars and Natural (bitvectors and integers with the underlying Why3) where we need to separate the part provable by provers with native support (CVC4 and Z3) from the part provable by using the theory (only Alt-Ergo from GNATprove).

Hence, the third and last asserts reformulate the postcondition for CVC4 and Z3 at the bit level. The three other assertions deal with conversions between modulars and Naturals, and are proved by Alt-Ergo. All together these assertions let us verify the specification of `PokeBit64`.

The results of the provers on `PeekBit8`, `PeekBit8Array` and `PokeBit64` are displayed in Figure 16, 17 and 18, respectively. We only describe the proof obligations that are not proved by all the provers:

- For `PeekBit8`, Figure 16, obligation 5 checks the postcondition of the function when `7 - Left` is smaller than `8`. This is the only obligation that is only proved by the provers with native support

Proof obligations		CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.0)	Z3 (4.4.0 noBV)	altergo (0.99.1)
VC for def	1. precondition	0.00	0.06	0.01	0.17	0.05
	2. precondition	0.01	0.03	0.01	0.01	0.05
	3. precondition	0.01	0.04	0.00	0.01	0.05
	4. VC for def	0.01	0.04	0.00	0.00	0.04
	5. postcondition	0.02	(5s)	0.02	(5s)	(5s)
	6. precondition	0.01	0.03	0.01	0.01	0.05
	7. VC for def	0.01	0.03	0.00	0.01	0.04
	8. postcondition	0.02	0.06	0.00	0.02	0.04

Figure 16: Provers results on PeekBit8

Proof obligations		CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.0)	Z3 (4.4.0 noBV)	altergo (0.99.1)
VC for def	1. precondition	0.02	0.05	0.01	0.01	0.06
	2. precondition	0.01	0.05	0.00	0.01	0.05
	3. assertion	0.02	0.11	0.02	(5s)	1.11
	4. precondition	0.02	0.08	0.04	0.20	0.10
	5. precondition	0.02	0.05	0.01	0.02	0.06
	6. postcondition	0.03	0.10	0.02	(5s)	0.06

Figure 17: Provers results on PeekBit8Array

for bitvectors. Obligation 8 checks the postcondition in the case where `7 - Left >= 8`. Indeed remember, from the second part of Section 4.1.2, that shifts are translated with a conditional in their second argument.

- For `PeekBit8Array`, Figure 17, obligation 3 checks that `Left / 8` is an index of `Addr`. The proof obligation 6 checks the postcondition of the function. There is no need for bitvector reasoning.
- For `PokeBit64`, Figure 18, the five obligations marked `assertion` are the translations of the five pragma assert in the function body. The last one translate the postconditon of the function. The two assertions at lines 22-23 and 26 are the ones that require bitvector reasoning.

Proof obligations		CVC4 (1.4)	CVC4 (1.4 noBV)	Z3 (4.4.0)	Z3 (4.4.0 noBV)	altergo (0.99.1)
VC for def	1. assertion	6.52	0.06	(5s)	0.78	0.06
	2. precondition	0.07	0.06	0.01	0.18	0.03
	3. assertion	(5s)	0.07	(5s)	(5s)	0.53
	4. precondition	0.12	0.06	0.02	0.19	0.06
	5. VC for def	0.02	0.03	0.01	0.00	0.05
	6. VC for def	0.02	0.03	0.00	0.00	0.05
	7. assertion	(5s)	4.30	0.29	(5s)	(5s)
	8. assertion	(5s)	0.09	(5s)	(5s)	1.22
	9. VC for def	0.02	0.03	0.00	0.00	0.05
	10. assertion	0.77	(5s)	0.07	(5s)	(5s)
	11. precondition	0.02	0.03	0.01	0.01	0.06
	12. VC for def	0.02	0.03	0.01	0.00	0.06
	13. VC for def	0.02	0.04	0.01	0.00	0.06
	14. VC for def	0.02	0.03	0.01	0.00	0.05
	15. postcondition	(5s)	0.08	(5s)	(5s)	(5s)

Figure 18: Provers results on PokeBit64

```
function MaxValue (Len : Natural) return Unsigned_64 is (Shift_Left (1, Len));

procedure Poke (Start, Len : Natural; Addr : in out Byte_Sequence;
                Value      : Unsigned_64; Result     : out Integer)
with
  Pre => Addr'First = 0 and then
         8 * Addr'Length <= Natural'Last and then
         Start + Len < Natural'Last and then
         Len in 0 .. 63,
  Post => (Result in -2 .. 0)
    and then
      ((Result = -1) = (Start + Len > 8 * Addr'Length))
    and then
      ((Result = -2) = (MaxValue (Len) <= Value and Start + Len <= 8 * Addr'Length))
    and then
      ((Result = 0) = (MaxValue (Len) > Value and Start + Len <= 8 * Addr'Length))
    and then
      (if Result = 0 then
         (for all I in 0 .. Start - 1 =>
                  Nth8_Stream (Addr'Old, I) = Nth8_Stream (Addr, I)))
    and then
      (if Result = 0 then
         (for all I in Start .. Start + Len - 1 =>
             Nth8_Stream (Addr, I) = Nth (Value, Len - I - 1 + Start )))
    and then
      (if Result = 0 then
         (for all I in Start + Len .. 8 * Addr'Length - 1=>
             Nth8_Stream (Addr, I) = Nth8_Stream (Addr'Old ,I)));
```

Figure 19: Specification of procedure `Poke`

4.2.3 The dual procedure `Poke`

A dual procedure of `Peek` called `Poke` amount to copy a 64-bit value into the stream. Its specification is given in Figure 19 it calls dual auxiliary functions. This code is specified and proved in a similar way as for `Peek`. The full annotated code is given in Section 4.3.

```
procedure PokeThenPeek (Start, Len : Natural; Addr : in out Byte_Sequence;
                        Value : Unsigned_64; Result : out Unsigned_64)
with Ghost,
  Pre =>
    Addr'First = 0 and then
    8 * Addr'Length < Natural'Last and then
    Len in 0 .. 63 and then
    Start + Len <= Addr'Length and then
    Value < MaxValue (Len),
  Post =>
    Result = Value;

procedure PokeThenPeek (Start, Len : Natural; Addr : in out Byte_Sequence;
                        Value : Unsigned_64; Result : out Unsigned_64)
is
  PokeResult : Integer;
begin
  pragma Assert (for all I in Len .. 63 => not Nth (Value, I));
  Poke (Start, Len, Addr, Value, PokeResult);
  pragma Assert (PokeResult = 0);
  Result := Peek (Start, Len ,Addr);
  pragma Assert (Eq (Result, Value));
end PokeThenPeek;
```

Figure 20: harness test `PokeThenPeek`

4.2.4 Proof Harness : Poke then Peek

In order to test our high-level specifications of `Peek` and `Poke`, we show how a simple test, that is to check whether the peek operation after a poke to the same part of the byte sequence, returns the same result as the input. The specification of `PokeThenPeek` and its body is given on Figure 20. The main goal, as given by the postcondition of `PokeThenPeek`, is to show that we get back, through `Peek`, exactly the same value we stored in the stream through `Poke`. To achieve the proof, we need to add three assertions in the body of `PokeThenPeek`, stating high-level intermediate properties. It is important to notice that there is nothing to specify at the bit level: the fact that Peek and Poke operate at the level of bits is hidden from their callers.

The results of the provers on `PokeThenPeek` are displayed in Figure 21. Notice that the first assertion takes more than one minute to be proved by CVC4. Indeed the proof is quite hard for provers as it mix integers, bitvectors (from the definition of `MaxValue (Len)`), as well as a quantifier in one go. In fact the dual function `PeekThenPoke` makes a similar assertion which needs a ghost function whose only purpose is to prove this assert.

Proof obligations		**CVC4 (1.4)**	**CVC4 (1.4 noBV)**	**Z3 (4.4.0)**	**Z3 (4.4.0 noBV)**	**altergo (0.99.1)**
VC for def	1. precondition	0.05	0.12	0.01	0.24	0.08
	2. VC for def	0.03	0.06	0.00	0.01	0.07
	3. assertion	63.07	(70s)	0.10	(70s)	(70s)
	4. precondition	0.08	0.10	0.01	0.22	0.08
	5. assertion	0.10	0.12	0.01	0.05	0.10
	6. precondition	0.17	0.13	0.02	(5s)	0.18
	7. VC for def	0.06	0.06	0.00	0.01	0.08
	8. assertion	0.86	0.18	0.03	(5s)	3.82
	9. postcondition	(5s)	0.14	0.06	(5s)	1.94

Figure 21: Provers results on PokeThenPeek

4.3 Full source code of BitWalker

The full file for the interface of the BitWalker package, containing all the specifications, is given below. The body of that package is then given afterwards.

bitwalker.ads

```
with Interfaces; use Interfaces;
with BitTypes;   use BitTypes;
with BitSpec;    use BitSpec;
with Bvext;      use Bvext;

package Bitwalker with
SPARK_Mode
is
   function PeekBit8 (Byte : Unsigned_8; Left : Natural) return Boolean is
      ((Byte and Shift_Left (1, 7 - Left)) /= 0)
   with
     Pre  => Left < 8,
     Post => PeekBit8'Result = Nth (Byte, 7 - Left);

   function PeekBit8Array (Addr : Byte_Sequence; Left : Natural) return Boolean is
     (PeekBit8 (Addr (Left / 8), Left rem 8))
   with
     Pre    => Addr'First = 0 and then
               Left < 8 * Addr'Length,
     Global => null,
     Post   => PeekBit8Array'Result = Nth8_Stream (Addr, Left);

   function PokeBit64
      (Value : Unsigned_64;
       Left  : Natural;
       Flag  : Boolean) return Unsigned_64
   with
     Pre    => Left < 64,
     Global => null,
     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 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,
       Global         => null,
       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 PeekBit64 (Value : Unsigned_64;
```

```
56                      Left  : Natural)
57                      return Boolean
58    is
59      ((Value and Shift_Left (1, 63 - Left)) /= 0)
60    with
61        Pre  => Left < 64,
62        Post => PeekBit64'Result = Nth (Value, (63 - Left));
63
64    function PokeBit8 (Byte : Unsigned_8;
65                       Left : Natural;
66                       Flag : Boolean)
67                       return Unsigned_8
68      with
69        Pre  => Left < 8,
70        Post =>
71          (for all I in 0 .. 7 =>
72             (if I /= 7 - Left then
73               Nth (PokeBit8'Result, I) = Nth (Byte, I))) and then
74          Nth (PokeBit8'Result, 7 - Left) = Flag;
75
76    procedure PokeBit8Array (Addr : in out Byte_Sequence;
77                             Left : Natural;
78                             Flag : Boolean)
79      with
80        Pre  => Addr'First = 0 and then Left < 8 * Addr'Length,
81        Post =>
82          (for all I in 0 .. 8 * Addr'Length - 1 =>
83             (if I /= Left then
84                    Nth8_Stream (Addr, I) = Nth8_Stream (Addr'Old, I)))
85           and then
86             Nth8_Stream (Addr, Left) = Flag;
87
88    procedure Poke (Start, Len : Natural;
89                    Addr       : in out Byte_Sequence;
90                    Value      : Unsigned_64;
91                    Result     : out Integer)
92      with
93        Pre  => Addr'First = 0 and then
94                8 * Addr'Length <= Natural'Last and then
95                Start + Len < Natural'Last and then
96                Len in 0 .. 63,
97        Post => (Result in -2 .. 0) and then
98                ((Result = -1) = (Start + Len > 8 * Addr'Length)) and then
99                ((Result = -2) = (MaxValue (Len) <= Value
100                                  and Start + Len <= 8 * Addr'Length))
101             and then
102               ((Result = 0) = (MaxValue (Len) > Value
103                                and Start + Len <= 8 * Addr'Length))
104             and then
105               (if Result = 0 then
106                  (for all I in 0 .. Start - 1 =>
107                         Nth8_Stream (Addr'Old, I) = Nth8_Stream (Addr, I)))
108             and then
109               (if Result = 0 then
110                  (for all I in Start .. Start + Len - 1 =>
111                         Nth8_Stream (Addr, I)
112                   = Nth (Value, Len - I - 1 + Start )))
113             and then
114               (if Result = 0 then
115                  (for all I in Start + Len .. 8 * Addr'Length - 1=>
116                         Nth8_Stream (Addr, I)
117                   = Nth8_Stream (Addr'Old ,I)));
```

```
118
119    function LemmaFunction (X : Unsigned_64; Len : Integer) return Unit
120      with
121        Ghost,
122        Pre  => Len in 0 .. 63 and then
123                (for all I in Len .. 63 => not Nth (X, I)),
124        Post => LemmaFunction'Result = Void and then
125                X < MaxValue (Len);
126
127    procedure PeekThenPoke (Start, Len : Natural;
128                            Addr       : in out Byte_Sequence;
129                            Result     : out Integer)
130      with
131        Ghost,
132        Pre =>
133          Addr'First = 0 and then
134          8 * Addr'Length <= Natural'Last and then
135          Len in 0 .. 63 and then
136          Start + Len <= 8 * Addr'Length,
137        Post =>
138            Result = 0 and then
139            (for all I in 0 .. 8 * Addr'Length - 1 =>
140               Nth8_Stream (Addr, I) = Nth8_Stream (Addr'Old, I));
141
142    procedure PokeThenPeek (Start, Len : Natural;
143                            Addr : in out Byte_Sequence;
144                            Value : Unsigned_64;
145                            Result : out Unsigned_64)
146      with
147        Ghost,
148        Pre =>
149          Addr'First = 0 and then
150          8 * Addr'Length < Natural'Last and then
151          Len in 0 .. 63 and then
152          Start + Len <= Addr'Length and then
153          Value < MaxValue (Len),
154        Post =>
155          Result = Value;
156
157 end Bitwalker;
```

end of `bitwalker.ads`

Here is now the body of the package.

`bitwalker.adb`

```
1  package body Bitwalker with
2    SPARK_Mode
3  is
4     ---------------
5     -- PokeBit64 --
6     ---------------
7
8     function PokeBit64
9        (Value : Unsigned_64;
10        Left  : Natural;
11        Flag  : Boolean) return Unsigned_64
12    is
13       Left_Bv : constant Unsigned_64 := Unsigned_64(Left);
14    begin
15       pragma Assert (Left_Bv < 64);
16       pragma Assert (63 - Left_Bv = Unsigned_64 (63 - Left));
```

```
17
18       declare
19          Mask : constant Unsigned_64 := Shift_Left (1, 63 - Left);
20          R : constant Unsigned_64 :=
21            (if Flag then (Value or Mask) else (Value and (not Mask)));
22       begin
23          pragma Assert (for all I in Unsigned_64
24                         range 0 .. 63 =>
25                           (if I /= 63 - Left_Bv then
26                               Nth_Bv (R, I) = Nth_Bv (Value, I)));
27
28          pragma Assert (for all I in Natural
29                         range 0 .. 63 =>
30                           (0 <= Unsigned_64 (I) and then
31                            Unsigned_64 (I) <= 63));
32
33          pragma Assert (Flag = Nth_Bv (R, 63 - Left_Bv));
34
35          return R;
36       end;
37    end PokeBit64;
38
39    ----------
40    -- Peek --
41    ----------
42
43    function Peek
44      (Start, Length : Natural;
45       Addr          : Byte_Sequence) return Unsigned_64 is
46    begin
47       if Start + Length > 8 * Addr'Length then
48          return 0;
49       end if;
50
51       declare
52          Retval : Unsigned_64 := 0;
53          Flag   : Boolean;
54       begin
55          for I in 0 .. Length - 1 loop
56             pragma Loop_Invariant
57               (for all J in Length - I .. Length - 1 =>
58                  Nth8_Stream (Addr, Start + Length - J - 1)
59                = Nth (Retval, J));
60
61             pragma Loop_Invariant
62               (for all J in Length .. 63 =>
63                  not Nth (Retval, J));
64
65             Flag := PeekBit8Array (Addr, Start + I);
66             Retval := PokeBit64 (Retval, (64 - Length) + I, Flag);
67          end loop;
68
69          return Retval;
70       end;
71    end Peek;
72
73    function PokeBit8 (Byte : Unsigned_8;
74                       Left : Natural;
75                       Flag : Boolean)
76                       return Unsigned_8
77    is
78       Mask : constant Unsigned_8 := Shift_Left (1, 7 - Left);
```

```
79    begin
80       return (if Flag then
81                  (Byte or Mask)
82               else
83                  (Byte and (not Mask)));
84    end PokeBit8;
85
86    procedure PokeBit8Array (Addr : in out Byte_Sequence;
87                             Left : Natural;
88                             Flag : Boolean)
89    is
90    begin
91       Addr (Left / 8) := PokeBit8 (Addr(Left / 8), Left rem 8, Flag);
92    end PokeBit8Array;
93
94    procedure Poke (Start, Len : Natural;
95                    Addr       : in out Byte_Sequence;
96                    Value      : Unsigned_64;
97                    Result     : out Integer)
98    is
99       Flag : Boolean;
100   begin
101      if Start + Len > Addr'Length * 8 then
102         Result := -1;
103         return;
104      elsif Value >= MaxValue (Len) then
105         Result := -2;
106         return;
107      end if;
108
109      for I in 0 .. Len - 1 loop
110         pragma Loop_Invariant (I in 0 .. Len);
111         pragma Loop_Invariant
112           (for all J in 0 .. Start - 1 =>
113              Nth8_Stream (Addr'Loop_Entry, J) = Nth8_Stream (Addr, J));
114         pragma Loop_Invariant
115           (for all J in Start .. Start + I - 1 =>
116              Nth8_Stream (Addr, J) = Nth (Value, Len - J - 1 + Start));
117         pragma Loop_Invariant
118           (for all J in Start + I .. 8 * Addr'Length - 1 =>
119              Nth8_Stream (Addr, J) = Nth8_Stream (Addr'Loop_Entry, J));
120
121         Flag := PeekBit64 (Value, (64 - Len) + I);
122
123         PokeBit8Array (Addr,
124                        Start + I,
125                        Flag);
126
127         pragma Assert (Nth8_Stream (Addr, Start + I)
128                        = Nth (Value, Len - I - 1));
129         pragma Assert
130           (for all K in Start .. Start + I - 1 =>
131              K /= Start + I and then
132            K in 0 .. 8 * Addr'Length - 1 and then
133            Nth8_Stream (Addr, K) = Nth (Value, Start + Len - K - 1));
134      end loop;
135
136      Result := 0;
137   end Poke;
138
139   function LemmaFunction (X : Unsigned_64; Len : Integer) return Unit is
140      Len_Bv : constant Unsigned_64 := Unsigned_64 (Len);
```

```
141    begin
142       pragma Assert (for all J in Len .. Len + (64 - Len) - 1 =>
143                         Nth (X, J) = Nth (Unsigned_64 (0), J));
144       pragma Assert (Eq_Sub (X, 0, Natural (Len_Bv), Natural (64 - Len_Bv)));
145       pragma Assert (Eq_Sub_Bv (X, 0, Len_Bv, 64 - Len_Bv));
146       pragma Assert ((X and (MaxValue (Len) - 1)) = X);
147       return Void;
148    end LemmaFunction;
149
150    procedure PeekThenPoke (Start, Len : Natural;
151                            Addr       : in out Byte_Sequence;
152                            Result     : out Integer)
153    is
154       Value : Unsigned_64;
155       AddrOld  : constant Byte_Sequence := Addr with Ghost;
156       V : Unit with Ghost;
157    begin
158       Value := Peek (Start, Len, Addr);
159
160       V := LemmaFunction (Value, Len);
161
162       Poke (Start, Len, Addr, Value, Result);
163
164       pragma Assert (Result = 0);
165
166       pragma Assert
167         (for all I in Start .. Start + Len - 1 =>
168            Nth8_Stream (Addr, I) = Nth8_Stream (AddrOld, I));
169    end PeekThenPoke;
170
171    procedure PokeThenPeek (Start, Len : Natural;
172                            Addr : in out Byte_Sequence;
173                            Value : Unsigned_64;
174                            Result : out Unsigned_64)
175    is
176       PokeResult : Integer;
177    begin
178       pragma Assert (for all I in Len .. 63 => not Nth (Value, I));
179
180       Poke (Start, Len, Addr, Value, PokeResult);
181
182       pragma Assert (PokeResult = 0);
183
184       Result := Peek (Start, Len ,Addr);
185
186       pragma Assert (Eq (Result, Value));
187    end PokeThenPeek;
188
189 end Bitwalker;
```

end of `bitwalker.adb`

4.4 Proofs

The proofs can be replayed with the command

```
gnatprove -P default.gpr --timeout=80 --prover=cvc4,z3,alt-ergo
```

The full SPARK analysis of the Bitwalker code is summarized below.

```
Summary of SPARK analysis
=========================

-----------------------------------------------------------------------------------------------------------------------
SPARK Analysis results        Total        Flow   Interval                                 Provers   Justified   Unproved
-----------------------------------------------------------------------------------------------------------------------
Data Dependencies                 .           .          .                                       .           .          .
Flow Dependencies                 .           .          .                                       .           .          .
Initialization                   23          23          .                                       .           .          .
Non-Aliasing                      .           .          .                                       .           .          .
Run-time Checks                  58           .          .                                58 (CVC4)          .          .
Assertions                       28           .          .    28 (altergo  14%, CVC4  75%, Z3  11%)          .          .
Functional Contracts             74           .          .      74 (altergo  3%, CVC4  95%, Z3  3%)          .          .
LSP Verification                  .           .          .                                       .           .          .
-----------------------------------------------------------------------------------------------------------------------
Total                           183    23 (13%)          .                               160 (87%)          .          .


Analyzed 4 units
in unit bitspec, 3 subprograms and packages out of 3 analyzed
  BitSpec at bitspec.ads:5 flow analyzed (0 errors and 0 warnings) and proved (0 checks)
  BitSpec.MaxValue at bitspec.ads:13 flow analyzed (0 errors and 0 warnings) and proved (0 checks)
  BitSpec.Nth8_Stream at bitspec.ads:9 flow analyzed (0 errors and 0 warnings) and proved (5 checks)
in unit bittypes, 1 subprograms and packages out of 1 analyzed
  BitTypes at bittypes.ads:3 flow analyzed (0 errors and 0 warnings) and proved (0 checks)
in unit bitwalker, 12 subprograms and packages out of 12 analyzed
  Bitwalker at bitwalker.ads:6 flow analyzed (0 errors and 0 warnings) and proved (0 checks)
  Bitwalker.LemmaFunction at bitwalker.ads:119 flow analyzed (0 errors and 0 warnings) and proved (18 checks)
  Bitwalker.Peek at bitwalker.ads:36 flow analyzed (0 errors and 0 warnings) and proved (26 checks)
  Bitwalker.PeekBit64 at bitwalker.ads:55 flow analyzed (0 errors and 0 warnings) and proved (4 checks)
  Bitwalker.PeekBit8 at bitwalker.ads:9 flow analyzed (0 errors and 0 warnings) and proved (4 checks)
  Bitwalker.PeekBit8Array at bitwalker.ads:15 flow analyzed (0 errors and 0 warnings) and proved (6 checks)
  Bitwalker.PeekThenPoke at bitwalker.ads:127 flow analyzed (0 errors and 0 warnings) and proved (12 checks)
  Bitwalker.Poke at bitwalker.ads:88 flow analyzed (0 errors and 0 warnings) and proved (45 checks)
  Bitwalker.PokeBit64 at bitwalker.ads:23 flow analyzed (0 errors and 0 warnings) and proved (15 checks)
  Bitwalker.PokeBit8 at bitwalker.ads:64 flow analyzed (0 errors and 0 warnings) and proved (6 checks)
  Bitwalker.PokeBit8Array at bitwalker.ads:76 flow analyzed (0 errors and 0 warnings) and proved (10 checks)
  Bitwalker.PokeThenPeek at bitwalker.ads:142 flow analyzed (0 errors and 0 warnings) and proved (9 checks)
in unit testwalker, 0 subprograms and packages out of 1 analyzed
  Testwalker at testwalker.adb:7 skipped
```

5 Conclusions

We designed a rich formal theory including arbitrary fixed-size bitvectors, a large set of bitwise operations, and a large set of operations involving both bitvectors 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 bitvector support or by other solvers that handle this theory as an axiomatic first-order theory. We presented several case studies illustrating how one can specify and prove a bit-level code correct with respect to a high-level specification.

To achieve these results, 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 a bit-level code satisfies a high-level post-condition, one may need to provide a hint under the form of an assertion rephrasing the post-condition at a level closer to the bits, and help the provers with assertions to enforce them to convert bitvectors to integers when required. Fortunately, as shown by proof harness poke-then-peek and peek-then-poke on the BitWalker, our approach allow a good modularity principle: as soon as a low-level code is given a high-level specification, the procedures calling such a code do not need to be aware that the low-level code operate at the bit level.

The support of Ada's modular types via bitvectors 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. In Section 4.1, we chose to map Ada's signed integer types to mathematical unbound integers. Another choice would be to map them to bit-vectors and use the signed arithmetic operators provided by SMTLIB. There are two reasons for choosing the first alternative:

- We made some experiments to encode signed types to bit-vectors, but we did not notice any improvement in the rate of automatically proved VCs, in fact we noticed regressions instead. In other words, the support for unbound integer arithmetic in SMT solvers is at least as good as the support for arithmetic operators of BV theory.
- Technically, checking that an arithmetic operation does not overflow when done on bitvectors is hard: since for example `bv_add` in SMTLIB silently wraps around in case of overflow, we can not check absence of overflow when computing `X+Y` by a simple formula like `(sle (bv_add X Y) #7FFFFFFF)`. The situation could become much different if BV theory of SMTLIB was providing operators that specifically detect overflows, as it was discussed recently on the SMTLIB mailing list.[17]

Related tools and experiments. The BitWalker case study was initially written in C and specified using the ACSL specification language of Frama-C. For that purpose a bitvector theory 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 bitvector theory to SMTLIB, we were able to prove this code fully automatically using SMT solvers. The input language, C versus Ada, is not important, although the choice between signed versus unsigned types in the source can make a difference in Ada: their semantics are significantly different.

Stefan Berghofer from Secunet company in Germany is already using the support for bitvectors in SPARK. He is using Isabelle/HOL to interactively discharge the VCs that cannot be proved automatically. It is applied in particular to big number package of libsparkcrypto[18] where there are a many occurrences of bitvector operations, in particular shifts.

[17] `http://www.cs.nyu.edu/pipermail/smt-lib/2015/000954.html`

[18] `https://bitbucket.org/sberghofer/libsparkcrypto/src/SPARK2014/src/shared/generic/lsc-bignum.adb`

Although there are a lot of applications of SMTLIB bitvector support to verification of hardware, we are not aware of any other case study of a formally proved software component involving bit-level operations.

Perspectives. Since Why3 is also used as a back-end for Frama-C, a short-term perspective is to integrate our bitvector support into Frama-C deductive verification plug-ins like Jessie or WP. Though, in C and unlike in Ada, there is the issue that it is not specified by the language whether overflow on signed or unsigned is allowed or not. In practice, it may be needed to provide some way to allow the user specify whether wrap-around semantics in case of overflow is intended or not. More generally speaking, there is a need to provide a way to attach to a machine word some abstract model, to specify the intended will of the programmer, e.g. as in the n-queens example when the programmer wants to interpret a machine word into the set of indices corresponding to the bits set to 1.

There is some need to apply the same approach to floating-point numbers, in order to exploit the new decision procedures for floating-point arithmetic that are now available in SMT solvers[19] [23]. In the past, floating-point programs were specified in terms of real numbers [8] and proved by specific solvers. As we did for bitvectors and integers, it is therefore desirable to design a theory that would allow combination of floating-point numbers with real numbers and at the same time can make use of SMTLIB support for floating-point arithmetic. Last but not least, there exists some code that operates on floating-point numbers at the bit-level [22]. Proving such a code would be a hard challenge, for example one may try to prove a program as complex as the tour-de-force fast inverse square root designed by Chris Lomont [20]:

```
float Q_rsqrt(float number)
{
  long i;
  float x2, y;

  x2 = number * 0.5F;
  y  = number;
  i  = *(long*)&y;
  i  = 0x5f3759df - (i>>1);    // evil floating point bit level hacking
  y  = *(float*)&i;
  y  = y*(1.5F - x2*y*y);
  return y;
}
```

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 bitvector theory. Thanks to Jean-Christophe Filliâtre for his comments on our modifications on his original code for the n-queens problem. Thanks to Yannick Moy for his comments on a preliminary version of this report.

[19] http://www.cprover.org/SMT-LIB-Float/

References

[1] John Barnes. *Programming in Ada 2012*. Cambridge University Press, 2014.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, pages 522–527. ACM, 1998.

[4] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[5] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[6] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[7] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also `http://toccata.lri.fr/gallery/fm2012comp.en.html`.

[8] Sylvie Boldo and Claude Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5:377–393, 2011.

[9] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *STTT*, 11(2):95–104, 2009.

[10] Roderick Chapman and Florian Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2014.

[11] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, number 7504 in Lecture Notes in Computer Science, pages 233–247. Springer, 2012.

[12] David Cyrluk, Harald Rueß, and Oliver Möller. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254, pages 60–71, Haifa, Israel, 1997. Springer-Verlag.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[14] Jean-Christophe Filliâtre. Verifying two lines of C with Why3: an exercise in program verification. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments (4th International Conference VSTTE)*, volume 7152 of *Lecture Notes in Computer Science*, pages 83–97, Philadelphia, USA, January 2012. Springer.

[15] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[16] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[17] Johannes Kanig, Edmond Schonberg, and Claire Dross. Hi-Lite: the convergence of compiler technology and program verification. In Ben Brosgol, Jeff Boleng, and S. Tucker Taft, editors, *Proceedings of the 2012 ACM Conference on High Integrity Language Technology, HILT '12*, pages 27–34, Boston, USA, 2012. ACM Press.

[18] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*. Springer, 2011. Materials available at `www.vscomp.org`.

[19] K. Rustan M. Leino and Valentin Wüstholz. The dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, volume 149 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–15, 2014.

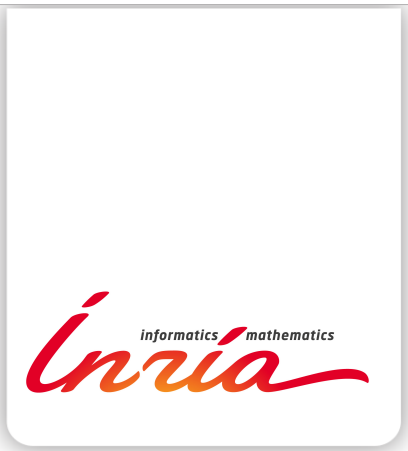
[20] Chris Lomont. Fast inverse square root. Technical report, Indiana: Purdue University, 2003. `http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf`.

[21] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[22] Thi Minh Tuyen Nguyen. *Taking architecture and compiler into account in formal proofs of numerical programs*. Thèse de doctorat, Université Paris-Sud, June 2012.

[23] Philipp Rümmer and Thomas Wahl. An smt-lib theory of binary floating-point arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.

[24] Henry S. Warren. *Hackers's Delight*. Addison-Wesley, 2003.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399