Why Hi-Lite Ada?

Jérôme Guitton, Johannes Kanig and Yannick Moy

AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France)
{guitton,kanig,moy}@adacore.com

Abstract. Use of formal methods in verification activities for critical software development is a promising solution to increase the level of confidence compared to the current practice based on testing, for increasingly complex programs, at a lower cost than the current approach. Concretely, the upcoming standard DO-178C for software development in avionics gives credit to formal verification for supporting verification activities. In project Hi-Lite, we pursue the integration of formal proofs with unit testing, for selected parts of a larger C or Ada software development. This integration relies crucially on a common language of specification between testing and formal proofs, where both share the same assertion semantics. For Ada, this language of specification based on subprogram contracts is part of the upcoming standardized version Ada 2012 of the language. In this paper, we describe the specifics of our translation from Ada to the intermediate verification language Why, noting which features of Why we used in our translation, and from which extensions of Why we could benefit in the future.

1 Introduction

Standards for critical software development define comprehensive verification objectives to guarantee the high levels of dependability we expect of life-critical and mission-critical software. All requirements must be shown to be satisfied by the software, which is a costly activity. In particular, verification of low-level requirements is usually demonstrated by developing unit tests, from which high levels of confidence are only obtained at a high cost. This is the driving force for the adoption of formal verification on an equal footing with testing to satisfy verification objectives. The upcoming DO-178C avionics standard states: *Formal methods [..] might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.*

Although there are some areas where formal verification can be applied independently [16], most areas where testing is the main source of evidence today would benefit from an integration of formal verification with existing testing practice. This combination should guarantee a coverage of atomic verifications through formal verification and testing. This is the goal of project Hi-Lite [12], a project aiming at combined unit testing and unit proof of C and Ada programs. The core feature enabling this combination is a language of specification (different in C and Ada) that can be both executed and formally analyzed. In this paper, we are interested only in the Ada part of Hi-Lite.

1.1 Contracts in Ada 2012

The forthcoming version of the Ada standard, called Ada 2012 [2], offers a variety of new features to express properties of programs. The most prominent of these are the *Pre* and *Post* aspects which define respectively the precondition and postcondition of a function[1]. These are defined as Boolean expressions over program variables and functions. Additionally, the expression in a postcondition can refer to the value returned by a function `F` as `F'Result`, and to the value in the pre-state (at the beginning of the call) of any variable `V` as `V'Old`.

The execution model for these aspects is simply to insert assertions at appropriate locations, which raise exceptions when violated. For each reference to the pre-state (`V'Old`), the compiler inserts a shallow copy of the corresponding variable's value at the beginning of the function body.

Expressing properties in contracts is greatly enhanced by the use of if-expressions, case-expressions, quantified-expressions and expression-functions, all defined in Ada 2012. The main objective of these *logic* features is verification by testing, based on their executable semantics. In particular, quantified-expressions are always expressed over finite ranges or (obviously finite) containers, with a loop through the range or container as execution model: `for all J in 0 .. 10 => P (J)` is true if-and-only-if the function `P` returns `True` for every argument, starting from 0 up to 10.

1.2 Assertion Based Verification

DO-178 [1] verification activities comprise notably requirement-based verification, which consists in checking that functions correctly implement the low-level requirements, and robustness verification, which consists in checking the absence of unintended behavior (like run-time errors) in normal and abnormal modes. As shown by work done at Airbus [16], low-level requirements can be mapped to formal contracts on functions, and requirement-based verification can be implemented with unit proofs, where function contracts are formally proved. Various projects using the platforms Frama-C and SPARK have also shown that absence of run-time errors can be proved formally on industrial programs [7, 15].

Both verification of function contracts and absence of run-time errors amount to checking assertions at the level of individual functions. In the following, we describe how to prove assertions, irrespective of where these assertions come from. A prerequisite to combining tests and proofs is both should yield consistent results, *i.e.,*, when a test fails, the corresponding assertion should not be possible to prove. This means that assertions should have the same dynamic and static semantics, as emphasized by Chalin [6].

1.3 Using Why to Generate Verification Conditions

Our method is based on the generation of verification conditions (VCs), *i.e.,* first-order formulas whose validity implies the correctness of the program w.r.t.

[1] Ada terminology uses the term *subprogram* for both functions and procedures. We use the term *function* instead, to ease the understanding of the paper.

its specification, via a weakest precondition calculus. To generate these VCs, we use the Why verification condition generator (VCG) as intermediate language.

Why is the VCG at the heart of the Why platform [10]. It features a small imperative programming language with a syntax similar to ML, and a polymorphic first order logic which serves as annotation language for programs.

Why provides minimal features such as side effects using references (but no aliasing), control structures such as if and while statements, and the possibility to define functions with annotations in the form of pre- and postconditions. It also features an exception mechanism and an effect system, which allows to limit the effect of a function or function declaration to a certain set of variables. Why enforces alias exclusion by allowing only one level of references, which excludes sharing, and by making use of the effect system to parameter aliasing.

From a user point of view, Why accepts a set of program functions as input, and outputs VCs, *i.e.,* These VCs can be generated in the syntax of many different automated provers, including Alt-Ergo[8].

Why programs are written in a syntax close to OCaml, but the part of the language used in this paper should be self-explanatory and mainly uses standard constructs such as assignments, sequences, loops, branchings and `let`-expressions. Logic annotations in programs are introduced with curly braces; curly braces at the beginning (the end) of a function introduce a precondition (a postcondition); curly braces in a loop introduce a loop invariant. Types can also carry annotations, such that the type

```
(x : int) -> { P } unit { Q }
```

denotes the type of functions which take an mathematical integers as only argument and return nothing (`unit` is the type whose only value is `void`), with precondition `P` and postcondition `Q`.

Ada is a case insensitive language, but an often-followed convention states that identifiers should start with a capital letter. On the contrary, Why identifiers must be lowercase. In this paper, we benefit from this syntactical difference to distinguish Why and Ada code; additionally, the encoding of an Ada type or variable in Why is its lowercase variant.

2 The Big Picture

2.1 Compilation Chain for Proofs

Project Hi-Lite aims at producing a set of tools and methodologies for applying formal verification to critical software development. GNATprove is the name of the formal verification tool of the Ada toolchain developed in Hi-Lite, which is based on the gnat2why translator from Ada to Why.

A main goal of the tools produced in Hi-Lite is to integrate easily in the usual project structure of Ada developments using the GNAT toolset. This means in particular being able to scale robustly to thousands of files grouped in hundreds of projects, physically hosted in various disks and network locations. Scalability in this context does not only mean to handle all these situations, but also to produce the expected outcome fast enough, including minimal recompilation after incremental changes to the source code. Robustness means that it should be possible to resume the process of generating and proving VCs correctly no matter what interruptions occur (process killed, network access lost, *etc.*).

The key to such a robust scalable process is to consider each phase leading to the proof of source assertions as a compilation phase, which could be applied in parallel to many different units. The goal of this special compilation is not semantic preservation, but conservative proof-checking: all assertions which should be proved to hold on the source program should correspond to assertion in the target code, and all assertions that are proved in the target code should hold in the source program *whenever a correct execution reaches the point of assertion.* Here is a sketch of the compilation phases:

$$\begin{tikzcd}
\text{Ada} \arrow[r, "\text{gnat}"'] \arrow[rr, bend left] & \text{ALI} \arrow[r, "\text{gnat2why}"'] & \text{Why Prog} \arrow[r, "\text{why}"'] & \text{Why VC} \arrow[r, "\text{alt-ergo}"'] & \text{Yes/No}
\end{tikzcd}$$

We start by running the GNAT compiler in a special mode to generate local effects in ALI (Ada Link Information) files from Ada source code. These effects list all variable reads, variable writes and function calls in a function. Then, we run the tool gnat2why, which translates an Ada unit into a set of Why program files, based on the information in ALI files. The Why VC generator then produces a set of VC files from the Why programs, and finally the SMT-prover Alt-Ergo attempts proving automatically these VCs. The tool called GNATprove is responsible for running the whole process through these different phases.

2.2 Proofs in Software Engineering

Another major goal of project Hi-Lite is to integrate formal proofs in the regular verification processes based on dynamic testing. It should be possible to apply proofs successfully to parts of a legacy codebase, without requiring a complete rewrite to follow predefined coding guidelines everywhere. We currently only consider sequential programs, although we could in the future consider supporting restricted forms of tasking, based on the Ravenscar profile for Ada [5].

We have defined a subset of Ada called Alfa, which excludes notably pointers (*access types* in Ada), implicit aliasing and exceptions. In particular, the absence of pointers greatly simplifies formal verification, by removing the need for a memory model, similar to the requirement in SPARK [3]. Notice that implicit aliases could still be present through parameter references. We also reject from Alfa code with such aliasing. These restrictions match the constraints of the critical software projects we target. We also provide a slightly modified version of the generic standard containers in Ada targeted at formal verification [9], which provide an attractive alternative to ad-hoc data structures based on pointers. New developments can use these so-called *formal* containers in Alfa code.

A function specification is in Alfa if its signature and contract are in Alfa. A function body is in Alfa if: (1) its specification is in Alfa; (2) its body only contains constructs that belong to Alfa; (3) its body only mentions variables whose type is in Alfa; (4) it only calls functions whose specification is in Alfa. Only functions whose body is in Alfa may be proved formally. This partition between functions not in Alfa, functions whose specification is in Alfa, and functions whose body is in Alfa, allows for a fine-grain application of formal proofs to selected functions. The tool GNATprove automatically detects to which category each function belongs, and it only applies the translation to Why to those specifications and bodies belonging to Alfa.

Note that local effects are generated for all functions, including those that are not in Alfa or whose body is not in Alfa. The tool gnat2why computes global effects for a function `S` by aggregating the local effects for `S` and all the functions in the downward call-graph closure of `S`, present in the ALI files of the corresponding units. Subprograms not in Alfa may contain reads and writes through pointers to data on the stack or the heap. These are abstracted as reads and writes to a special `Heap` variable. Regarding global variables, only direct reads and writes, as well as reads and writes through references, are considered. All other cases of reads and writes to variables (through a memory address or through a call to a function pointer) are forbidden in the programs we consider.

Note also that, in order to prove function `S` (in Alfa), we require that all bodies of functions in the downward call-graph closure of `S` are available (even those of functions not in Alfa). This requirement is similar to the one for testing `S` by execution. This is a strong limitation, but it still allows applying formal proofs incrementally on an existing codebase.

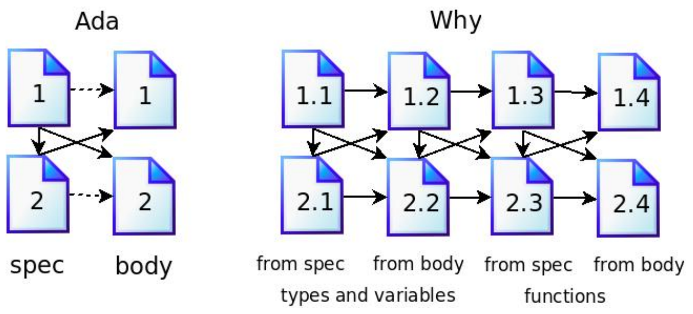
2.3 Ada to Why Mapping

In order to translate Ada code to Why, we had to deal with two major issues: first, Ada units can be mutually dependent on each other, while the graph of dependencies between Why files should form a directed acyclic graph (DAG); secondly, the source units we consider contain both functions that can be translated to Why, and functions that cannot be translated to Why.

The unit dependency issue means that an Ada unit cannot be simply translated into a Why file. One could consider translating separately the specification and implementation of an Ada unit into two different Why files: function specifications would be translated into Why function declarations and function bodies into Why *functions*, with the same Why contract for the function declaration and the function (including global effects) in order to ensure correctness of the approach. This does not work either because the global effects generated for a function declared in the specification may have to mention a global variable declared in the body. Overall, we had to take into account both intra-unit dependencies and inter-unit dependencies between types, variables and functions. Note that simple and mutual recursion between functions is just one of these dependencies. In the end, we chose to translate an Ada unit into four Why files:

1. a file for types and variables from the Ada specification;
2. a file for types and variables from the Ada body;
3. a file for function declarations from the unit;
4. a file for function bodies from the unit.

The dependencies between these generated files form a DAG, as depicted below:



Dependencies of a file at level $i \in 2..4$ on a file at a lower level do not introduce circularities; dependencies of a file at level 1 or 3 on a file at the same level follow the non-circular dependencies between Ada unit specifications.

Subprograms whose specification is not in Alfa are not translated into Why. For functions whose specification is in Alfa but whose body is not in Alfa, only the function specification is translated into Why.

3 Ada to Why Translation

For the most part, our translation from Ada to Why is similar to existing translations from C or Java to Why [10]. The usual statements (branchings, loops, assignments) and expressions (including the new Ada 2012 expressions) are translated to the corresponding expressions in Why. Arrays are translated as abstract types axiomatized with the theory of functional arrays. Records are translated as abstract types axiomatized with the theory of tuples. Enumerations are translated like integer types. In this section, we take a closer look at the translation of Ada integer types and assertions from contracts and loop invariants.

3.1 Essential Features Of Why

There are a number of features of Why that greatly simplify our task of generating correct Why code. The first one is the functional nature of Why, namely the fact that expressions and statements are *not* separated, but can be freely mixed. This is specially handy when translating those nodes of the GNAT compiler internal tree for Ada expressions which contain statements. It is also convenient to be able to introduce local variables at any time, for example to name intermediate results that should be computed only once.

Why provides a powerful exception mechanism, integrated into the effect system, which allows not only to raise and handle exceptions, but also to specify what a function guarantees in the case of a raised exception. We heavily rely on exceptions, notably to translate loops (see also Section 3.3) and to encode functions with multiple return statements.

The demonic choice operator or any-operator, written `[type]`, returns an arbitrary value of a given type. This is very useful in many situations, for example to translate Ada 2012 quantified expressions (see Section 3.3).

Why allows to introduce labels in programs, denoting program points of interest. In annotations, one can then refer to the value of mutable variables at that point. We plan to use this feature to allow the user to refer to the loop entry point in a loop invariant, for example.

Another crucial feature of Why, of more practical nature, is the ability to track source locations. Each expression and formula of a Why program can be given a name that is associated to a source location of the Ada source program. When generating VCs, Why can now associate each VC to the original Ada source location, and inform about the kind of VC. The possible kinds include preconditions, assertions, and loop invariants. This feature is essential for the generation of helpful error messages.

We do not currently use the type polymorphism provided by Why, with the exception of the `ignore` function introduced in Section 3.3.

3.2 Ada Integer Types, Range and Overflow

The Ada standard defines a few predefined integer types such as `Integer`, `Long_Integer`, *etc.* The exact range of these predefined types depends both on the compiler and the target platform. Usually, they correspond to machine integer types of the underlying architecture, such as 32-bits or 64-bits integers. GNATprove uses the values defined by the GNAT compiler for a given target.

Additionally, the programmer can define his own integer types, using the syntax

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

or *derived types* from existing types, using the syntax

```
type One_Ten_Derived is new Integer range 1 .. 10;
```

or *subtypes* of existing types, using the syntax

```
subtype One_Ten_Integer is Integer range 1 .. 10;
```

The two types `One_Ten` and `One_Ten_Derived` are *fresh* types, different from predefined types such as `Integer`; functions can be overloaded for these types (*e.g.,* arithmetic operations), and an explicit conversion is needed to transform a value between these types. The subtype `One_Ten_Integer` does not introduce a new type, so values of type `One_Ten_Integer` are really of type `Integer`.

The three types above have the same range 1 to 10, meaning that a value of such a type should always belong to this range. Any attempt to store a too small or too large value results in a range check failure at run time. This is not to confound with overflow checks, controlled by the *base type*. Each integer type has a corresponding base type, which is the type of intermediate results in an arithmetic expression. Any computation of a too small or too large intermediate value results in an overflow check failure at run time. Both `One_Ten_Derived` and `One_Ten_Integer` share the same base type as `Integer`, which is defined as `Integer` itself. The base type of `One_Ten` depends on the compiler and target, with a minimal range guaranteed of -10 .. 10.

Consider three variables `X`, `Y` and `Z` of type `One_Ten`. Then the assignment

```
Z := (X + Y) / 2;
```

may raise an overflow error, because the sum of `X` and `Y` may exceed the range of the base type, but not a range error, because the overall result of the right hand side will always be in the required range. On the other hand, if `X`, `Y` and `Z` are of type `One_Ten_Derived` or `One_Ten_Integer`, then the assignment

```
Z := X + Y;
```

may not raise an overflow error, because the range of the base type `Integer` is used to carry out the computation. However, if the sum exceeds 10, then the assignment will result in a range error.

Despite the complexity of Ada integer types, we were able encode them into Why like done by Jean-Christophe Filliâtre in the Caduceus tool [10], as shown in Fig. 1. This theory contains conversion functions from and to the type `int` of mathematical integers, a range predicate, and a number of axioms. It is worth noting that the *logical* conversion functions are considered to be total, while the conversion functions, to be used in the Why program space, have a precondition which limits their use to mathematical integers that are in the range of that type. All values of the Ada integer types are considered to be in the range (axiom `t_range`), and conversion back and forth can be eliminated (axiom `t_coerce`). The function declaration `t_in_range_` is the identity function on mathematical integers, but its argument must be in the range of type `t`; this allows a seamless insertion of overflow checks in arithmetic expressions.

All arithmetic operations are translated to Why using the operations on mathematical integers. Conversions and overflow checks are added as needed and may trigger verification conditions. As an example, assume again the variables `X`, `Y` and `Z` to be of type `One_Ten_Derived`, and the assignment

```
Z := X + Y;
```

Then the corresponding code in Why looks as follows:

```
z := one_ten_derived_of_int_ (integer_in_range_
       (one_ten_derived_to_int (x) + one_ten_to_int (y))));
```

The function calls with the suffix `_` generate VCs corresponding to range and overflow checks in Ada.

```
type t
predicate t_in_range (x : int) = -128 <= x <= 127

logic t_to_int : t -> int
logic t_of_int : int -> t
parameter t_of_int_ :
   n : int -> { t_in_range (n) } t { t_to_int (result) = n }
parameter t_in_range_ :
   n : int -> { t_in_range (n) } int { result = n }

axiom t_range :
   forall x : t.  t_in_range (t_to_int (x))
axiom t_coerce :
 forall x : int.  t_in_range (x) -> t_to_int (t_of_int (x)) = x
axiom t_unicity :
 forall x, y : t.  t_to_int (x) = t_to_int (y) -> x = y
```

Fig. 1. The theory in Why for an Ada integer type of the range `-128..127`.

3.3 Executable Semantics for Assertions

Assigning the same semantics to assertions in tests and in proofs leads to some differences between our translation and the usual approach, in which assertions are considered as predicates. In particular, we must take into account possible run-time errors during the evaluation of an assertion.

Guarded assertions. Consider a function `Add` with the following specification:

```
function Add (X, Y : One_Ten) return One_Ten
  with Pre (X + Y < 10);
```

In the following discussion, let us focus on the precondition of the function `Add`. In classical logic, this assertion can be true or false, depending on the assumptions that are available concerning `X` and `Y`. However, if we consider it as an assertion in a program, there are three possible outcomes: the expression can evaluate to one of the Boolean values true or false, but the addition `X+Y` can also overflow, which, in Ada, will trigger an exception.

There are several possibilities to deal with this mismatch. One option is to assimilate failing assertions to false assertions, so that the above assertion not only states that the sum of `X` and `Y` is less than `10`, but also that this sum does not overflow nor underflow. This semantics can be realized by adding these additional assertions to formulas when translating them to logic assertions. Such implicit assertions include both run-time checks and preconditions of functions that are used in specifications. This approach, implementing the *run-time assertion semantics* of formulas, has been chosen for example in ESC/Java2.

In Hi-Lite, we decided not to follow this option, because the presence of implicit assertions that are not explicitly stated in the program text is not suitable in the context of critical software development. Instead, we require the programmer to write guarded assertions, *i.e.,* assertions that cannot raise a run-time error, so that arithmetic operations, array accesses, *etc.*, must be protected in assertions. We check the presence of these guards by generating VCs to prove the absence of run-time errors in assertions. Preconditions must be run-time error free in any context, but local assertions and postconditions can use information that stems from the enclosing context. Note that the Dafny system [13, 11] implements the same behavior.

In practice, we generate such VCs by translating assertions like programs, without needing a special mechanism such as the `is-defined` operator of Chalin [6]. The translation of the function `Add` looks as follows:

```
let add (x : one_ten) (y : one_ten) =
   { true } (* make no assumption at all *)
   ignore
     (one_ten_range_
       (one_ten_to_int (x) + one_ten_to_int (y)) < 10);
   assume { one_ten_to_int (x) + one_ten_to_int (y) < 10 };
   ... (* translated body of function Add *)
```

Note the use of the `assume` statement and the `ignore` function, which can be declared in Why as follows, using polymorphism:

```
logic ignore : 'a -> unit
```

Finally, note that in all generated VCs, `x` and `y` are assumed to be in the range of the type `one_ten`, as stated by the axiom `one_ten_range`.

In this particular example, the overflow check cannot be proved, because the addition can overflow, and GNATprove will report an unproved VC.[2] This example illustrates the major difference between our approach and the one chosen in ESC/Java2 [6]: in our model, the formulas which express that assertions are free from run-time errors always appear as *assertions* in Why, *i.e.,* as formulas to be proved, never as *assumptions.* So, the precondition of the `Add` function is incorrect in GNATprove, while on the contrary it would be *stronger* than the one written by the user in ESC/Java2.

Evaluation order. In logic annotations, the order of evaluation is not important, as logic functions cannot have side effects or provoke an error. However, to reflect the executable semantics, a few precautions have to be taken. The first issue is that, whenever the exact order of evaluation is not defined, it should not influence the result of an expression. Such situations arise for example when evaluating the parameters of a function call. In Hi-Lite, we guarantee that by disallowing global effects in Ada functions (as opposed to Ada procedures), so that function calls in the same expression cannot interfere. A more precise analysis

[2] This VC will be located at the precondition at the function, and it does not depend on this function being called elsewhere in the program.

would of course be possible. The second issue is to correctly reflect the evaluation order, when defined. As a prominent example, the shortcut operator `and then`, which corresponds to the `&&` operator in C is translated to an if-then-else statement in programs, and to a simple conjunction in formulas.

Quantified Expressions The semantics of quantified expressions (see Section 1.1) is sufficiently complex to raise a number of questions with regard to formal verification. They act as loops over an integer range or the contents of a container, but they exit early, as soon as the Boolean result is determined. Consider the following pathological example:

```
(for all J in 1 .. 10 => (if J = 5 then J /= 1 / 0 else False))
```

This always returns `False`; The potential run-time error at the fifth iteration is never attained, because already the first iteration yields `False`.

For the proofs, we opt for a stricter, clearer semantics by fixing that the enclosed expression must not raise a run-time error, given *any* loop index in the range. This means that the pathological example above is not provable in GNATprove because we require that the quantified expression does not raise an exception for the entire range.

To that effect, we generate the following code, using the any-operator twice:

```
ignore (let j = [ { } type { range (result) } ] in expr);
[ { } bool { if result then forall (i:type). expr_as_pred } ]
```

Why generates the required checks on the quantified expression and at the same time expresses the contents of the Boolean result, using first-order quantification in Why. Note that the Ada expression is translated twice, once to a Why program expression to obtain the checks, and once to a Why predicate.

Two reviewers of an earlier version of this paper pointed out that it is possible and easy to model the more complex semantics, by adding the additional hypotheses that previous loop iterations did not fail, and returned `True` (in the case of a universal quantification). We still like the simplicity of the current approach, but may change in the future.

Loop assertions. Loop invariants are necessary to prove interesting properties of code containing loops; gnat2why interprets assertions at the beginning of a loop as loop invariants. A naive translation would transform the Ada code

```
while C loop
   pragma Assert (P);
   ...
end loop;
```

into a Why loop whose invariant is simply the predicate `p1`. But such a translation would be incorrect because it does not match the dynamic semantics of assertions. Indeed, loop invariants in Why follow the usual semantics à la Hoare:

the invariant must initially hold, regardless of whether the loop executes at all, and it must hold when exiting the loop. In order to match the dynamic semantics of the Ada program including the assertion, then it should suffice that the assertion holds at the beginning of each loop iteration. This semantics can be achieved in Why using the following translation scheme:

```
if c then
   try
      while true do (* infinite loop *)
         { invariant c and p}
         ...
         if not c then raise Exit
      done
   with Exit -> ();
```

Here, we replace the bounded loop by in infinite loop that exits using an exception. We also protect the loop using an if statement, so that the invariant does not need to hold if the loop is not executed, and we use a `raise` statement to exit the loop, so that the invariant does not need to hold when the loop terminates. Finally, we add the condition `c` to the loop invariant, so that this information is present in VCs generated for the loop body.

In general, loop invariants following this *assertion semantics* are weaker than loop invariants following Hoare semantics, because they need to hold at fewer execution points; as a consequence, they are sometimes slightly easier to write. A potential drawback of assertion semantics is that the VC generator cannot simply forget about the loop body for the rest of the code, but has to check the different possibilities to exit the loop. Why deals nicely with this situation.

4 Conclusion and Future Work

In this paper, we presented a translation from Ada to Why, in a context where only parts of a program are proved, and assertions are assigned the same semantics in formal verification as during execution. Various constructs of the Why language were essential in making this translation easier: functional orientation, demonic choice, exceptions and source location tracking all simplify the work which has to be done to translate Ada to Why.

The next generation of the Why tool, called Why3 [4], provides many features that we believe make it a superior choice as a backend of our GNATprove technology. We are currently preparing a migration to Why3 in the near future.

In particular, the Why3 feature of *theories* will allow much of the Why code that is currently generated programmatically (for example the theory of finite integer types) to be written once and for all in a theory, which can then be *cloned* and specialized. This would simplify the code generator, but also the readability of the generated code. This feature seems to be more powerful than the parametric polymorphism existing in the current version of Why, although both serve the same purpose of writing modular specifications.

Future Work on GNATprove consists in adding features of Ada to the supported Alfa language, and providing means to help the programmer write correct programs and meaningful specifications, such as avoiding trivial assertions, obviously false preconditions, redundant assertions and incomplete contracts.

A challenge is the support of nested arrays and records [14]. Even though modeling records is slightly simpler in a setting without aliasing, where record fields can be modeled by references, this simple approach breaks down when arrays can contain records. One currently envisioned solution is to model arrays of records by records of array fields, and to do so recursively, if necessary.

References

1. DO-178B: Software considerations in airborne systems and equipment certification, 1982.
2. `http://www.ada-auth.org/standards/12rm/html/RM-TTL.html`.
3. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
4. F. Bobot, J.-C. Filliâtre, A. Paskevich, and C. Marché. Why3: Shepherd your herd of provers. Unpublished, 2011.
5. A. Burns. The Ravenscar profile. Technical report, University of York, 2002. Available at `http://www.cs.york.ac.uk/~burns/ravenscar.ps`.
6. P. Chalin. Engineering a sound assertion semantics for the verifying compiler. *IEEE Trans. Softw. Eng.*, 36:275–287, March 2010.
7. R. Chapman. Industrial experience with SPARK. *SIGAda Ada Letters*, Dec. 2000.
8. S. Conchon and E. Contejean. The Alt-Ergo automatic theorem prover, 2008. `http://alt-ergo.lri.fr/`.
9. C. Dross, J.-C. Filliâtre, and Y. Moy. Correct Code Containing Containers. In *5th International Conference on Tests & Proofs (TAP'11)*, Zurich, June 2011.
10. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590, pages 173–177, Berlin, Germany, July 2007.
11. J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01a, University of Central Florida, School of EECS, 2009.
12. `http://www.open-do.org/projects/hi-lite/`.
13. K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
14. D. C. Luckham and N. Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Trans. Program. Lang. Syst.*, 1:226–244, October 1979.
15. D. Pariente and E. Ledinot. Formal verification of industrial C code using Frama-C: a case study. *Formal Verification of Object-Oriented Software*, June 2010.
16. J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 532–546, Berlin, Heidelberg, 2009. Springer-Verlag.