Hi-Lite: The Convergence of Compiler Technology and Program Verification

Johannes Kanig
AdaCore
46 rue d'Amsterdam
Paris, France
kanig@adacore.com

Edmond Schonberg
AdaCore
104 Fifth Avenue
New York, NY
schonberg@adacore.com

Claire Dross
AdaCore
46 rue d'Amsterdam
Paris, France
dross@adacore.com

ABSTRACT

Formal program verification tools check that a program correctly implements its specification. Existing specification languages for well-known programming languages (Ada, C, Java, C#) have been developed independently from the programming language to which they apply. As a result, specifications are expressed separately from the code, typically as stylized comments, and the verification tools often bear no direct relation to the production compiler. We argue that this approach is problematic, and that the compiler and the verification tools should be integrated seamlessly. Based on our current work on the Hi-Lite project to develop a formal verification tool for Ada 2012, we show that in an integrated setting, the compiler becomes the centerpiece of the verification architecture, and supports both static proofs and run-time assertion checking. Such an environment does much to simplify software certification.

Categories and Subject Descriptors

F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*Pre- and Postconditions, Mechanical Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods, Programming by Contract*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

Keywords

Compiler technology, formal verification, testing

1. INTRODUCTION

Most programming languages commonly used in embedded or critical software do not have suitable specification features for formal verification, such as pre- and postcondition for subprograms. Hence, such features end up being developed as independent specification languages for Ada [3], C [12, 8, 11], Java [6] and C# [4]. Since the compilers do not process these specifications, they are typically embedded in the source programs as special comments (SPARK, JML, E-ACSL), or as arguments to preprocessor directives that evaluate to the empty string (VCC, eCv). Only the authors of Spec# have extended the C# language with specifications and created a new compiler to handle code and specifications, different from the Microsoft production compiler, which causes some problems we discuss in Section 3. The verification tools understand specifications, and they can parse together the code and the specifications to create an annotated syntax tree for the program, on which proof machinery can operate.

A direct consequence of departing from the official language definition – by extending the language or by interpreting comments or arguments of preprocessing directives – is that the compiler and the verification tools cannot share the same frontend. Said otherwise, the existing compiler technology cannot be used directly as a frontend for these verification tools. There are also many reasons for not modifying an existing compiler to support these specifications: a compiler may not be readily available (the situation when SPARK was born) or modifying the internals of the compiler is considered too difficult (this is the reputation of gcc).

With the advent of Ada 2012, the latest version of the language, the situation is quite different, because contract features are now in the official language definition, so that any Ada 2012 compiler must support them. In this paper, we argue that not only *can* a compiler and formal verification share the same frontend, but they *should*. We discuss the advantages and disadvantages that this choice implies. We illustrate our case using the GNAT compiler and the verification tools GNATtest and GNATprove, which are developed in the context of the research project Hi-Lite [17].

In Section 2, we describe the new assertion mechanisms of Ada 2012, and we show how important it is that these assertions be built-in into the language, from a user point of view. In Section 3, we show how important this is for the tool developer, and we discuss in more detail the central role the compiler can play in a formal verification toolchain. In Section 4, we discuss related issues such as providing proved libraries and dealing with mathematical features in specifications.

2. EXECUTABLE ASSERTIONS AND CONTRACTS

The relationship between programming language design and correctness proof methodologies is an unsettled one. The pioneering articles of Floyd and Hoare use annotations

HILT'12, December 2–6, 2012, Boston, Massachusetts, USA.

that are added to existing code, without suggesting any particular relationship between the language of annotations and the surrounding programs. The methodology is aimed at manual proofs, and given the relative lack of abstractions in the programming languages of the time, the assertions are mostly arithmetic relations on integer values. Since then, all kinds of different logics (e.g. separation logic [22]) have been applied to all kinds of programming languages, such as C.

By contrast, the work of Dijkstra's school sees program development as evolving from pre/postconditions to code, and in fact in [13] most of the work consists in defining rigorously the logical formalism to use in the assertion language, and its relation to the eventual programs. However, there is no indication that authors envision some compilation machinery to relate assertions and code.

The development of Eiffel [1] takes a much more pragmatic approach. No doubt influenced by the slow progress in automated proof technologies, and the lack of enthusiasm on the part of the programming community for the rigors of Dijkstra's approach, Eiffel proposes that assertions about the behavior of code are (in addition to an expression of logical intent) optional run-time checks to be inserted in the code. As such, assertions and code are written in the same programming language. Meyer is also keenly aware that proof technology has matured sufficiently to be applied to contemporary software, and in [20] describes the use of Eiffel assertions in automated tools. A similar approach that combines formal verification and simulation applied to hardware design has been followed in the System Verilog community [24].

Ada 2012 follows the same two-pronged approach: the language offers a rich set of assertion mechanisms, where assertions are simply boolean expressions in Ada itself, that can be translated into run-time checks, and used profitably by automated tools to check whether these assertions are in fact obeyed by the code. The following hybrid approach then suggests itself: assertions that can be proved by static analysis can be removed from the code; those that can be proved to be false can be used to generate counterexamples; the remaining ones can generate run-time checks that will trigger an appropriate exception when violated.

To support a richer language of assertions, Ada 2012 incorporates a number of new expression forms, that allow a more functional style of programming than was the case for previous versions of the language.

2.1 New Expression Forms

Assertions can be used to specify the desired behavior of subprograms, loops, object creation routines, etc. In all these cases, the assertions are predicates, that is to say expressions that yield boolean values. To enrich the syntax of assertions, Ada 2012 offers several new boolean constructs. Some of them are familiar from other languages, and do not need much discussion: conditional expressions (if-expressions and case-expressions) and expression functions are similar to the constructs found in functional languages. Expression functions simplify immensely the introduction of abstractions in assertions.

Quantified expressions are familiar constructs in Logic, but are only rarely found in programming languages. They use familiar syntax:

```
for quantifier loop_parameter_specification
    => predicate
```

For example, the following expresses that array A is sorted:

```
for all I in A'Range =>
    I = A'Last or else A(I) <= A(I'Succ)
```

and the following states (inefficiently) that a number N is composite:

```
for some I in 2 .. N / 2 => N mod I = 0
```

It should be noted that quantified expressions are always expressed over a finite discrete range. It is up to the user to be careful no to choose a range that is too large and whose execution will take a very long time.

2.2 Contracts

Contracts specify the expected behavior of various language constructs. Even though the effect of any contract could be specified by using a single assertion mechanism, it is convenient to provide several varieties of contracts, depending on whether they apply to the behavior of subprograms or to properties of objects. In all cases, contracts are specified as *aspects* of entities. Aspects generalize the earlier Ada notion of attribute specification. Aspect specifications are attached directly to the declaration of the entity to which they apply.

2.2.1 Pre- and Postconditions

Predicates that express requirements on the input parameters of a function, and those that specify the characteristics of the result, are the most familiar. For example, a square root routine may be given the following declaration:

```
function Sqrt (X : Float) return Float
with
  Pre => X >= 0.0;
```

Postconditions often need to refer to the original value of an actual, and to the result of a function. The attributes `'Old` and `'Result` fulfill this need. We may add the following to our `Sqrt` routine:

```
function Sqrt (X : Float) return Float
with
  Pre  => X >= 0.0,
  Post =>
    ((Sqrt'Result) ** 2 <= X
     and then (Sqrt'Result + Epsilon) ** 2 > X);
```

where Epsilon is a constant declared elsewhere.

In a language with inheritance and polymorphism, we need to specify how the operations of a type T are inherited by extensions of T. In Ada 2012, the aspects `Pre'Class` and `Post'Class` of a subprogram P are inherited, *i.e.,* they also apply to an overriding of operation P on a descendant of T. These inheritance rules are consistent with the well-known Liskov substitution rules.

2.2.2 Type Invariants

In an imperative language with mutable objects, it is important to be able to define consistency conditions on an object of some composite type T. Whenever an object of this type is created or modified, we want to verify that the

stated invariant is respected. However, if the type has visible components, some of which may be by-reference types, it is infeasible to verify that the invariant is respected after each potential modification. Therefore Ada 2012 chooses to allow invariants only on private (*i.e.,* opaque) types, in which case the invariant is typically expressed as a function call. The invariant must be checked whenever a type constructor is invoked, and whenever a subprogram that is visible to a client modifies an object of the type.

Here again inheritance and polymorphism require a separate mechanism to separate type-specific invariants from invariants that are intended to apply to all members of a class of types. The invariant `Type_Invariant'Class` indicates a property that all objects of a type descended from a given type must obey.

2.2.3 Subtype Predicates

In contrast with type invariants, subtype predicates typically apply to types with visible characteristics. Their simplest use is to provide for subsets of scalar types whose values are not contiguous. For example:

```
subtype Multiple is Natural with
  Dynamic_Predicate => Multiple mod 3 = 0;
```

If the bounds of a scalar subtype are static, and a `Static_Predicate` applies to it, the subtype can be used as the domain of iteration of a loop. On the other hand, if the bounds are non-static and the subtype has a `Dynamic_Predicate` defined, it cannot be used in a loop. If a subtype has a predicate of either kind, it cannot be used as the index of an array, and the attribute `'First`, `'Last`, or `'Range` cannot be applied to objects of the subtype. The subtype predicate is evaluated at places where a conversion to the subtype takes place: initialization, assignment, parameter passing.

2.3 The Advantages of Executable Contracts

The possibility of making assertions and contracts part of the executable benefits the programmer in two ways:

- it gives the programmer a gentle introduction to the use of contracts, and encourages him to develop assertions and code in parallel. This is natural when both are expressed in the same programming language.
- executable assertions can be enabled and checked at run time, and this gives valuable information to the user. When an assertion fails, it means that the code failed to obey desired properties (*i.e.,* the code is erroneous), or that the intent of the code has been incorrectly expressed (*i.e.,* the assertion is erroneous) – and experience shows that both situations arise equally often. In any case, the understanding of the code and properties of the programmer are improved. This also means that users get immediate benefits from writing additional assertions and contracts, which greatly encourages the adoption of contract-based programming.
- contracts can be written and dynamically verified even when the contracts or the program are too complex for automatic proof. This includes programs that explicitly manipulate pointers, for example.

Executable contracts can be less expressive, or more difficult to write, in some situations. This is discussed in Section 4.

In summary, Ada 2012 in itself enables contract-based, dynamic verification of complex properties of a program. GNATprove enables contract-based static deductive verification of a large subset of Ada 2012.

3. SHARING THE COMPILER FRONTEND

We have discussed in the introduction how, given the absence of annotations in some programming languages, program verification tools cannot use an existing compiler frontend for their syntactic and semantic analysis phase, and instead must develop their own parser/semantic analyzer for the language, enriched with a similar tool for the annotations. We also have indicated that the situation is different in a language whose definition includes annotations, such as Ada 2012. We use the GNAT frontend to parse and analyze Ada programs, and take the GNAT semantically analyzed syntax tree as input to the various verification tools we develop.

3.1 A Program Verifier as Backend of the Compiler

The first obvious advantage of using an existing frontend is that you do not have to write your own. Even for moderately complex languages such as C, this is a major undertaking[1], and for more complex ones such as Ada it is simply prohibitive. Complex issues such as scoping, overload resolution, generic instantiations, etc. have to be handled before any verification activities can proceed. It is clear that the workaround that consists in restricting the verification tool to a subset of the language, for which an independent frontend becomes feasible (as was done by Praxis for SPARK), is still a major investment. Note that GNATprove also applies to a subset of the full Ada 2012 language, which notably excludes pointers, exceptions and side effects in expressions (and contracts), but this subsetting is solely motivated by the limitations of the proof tools, and not because of frontend considerations.

Even disregarding the initial amount of work that has to be put in to obtain a parser and semantic analyzer for an existing language, this tool now must be maintained and updated when a new language version comes out, or when bugfixes have been applied to the original tool. The authors of the Spec# verifier have experienced this maintenance problem, which led them to adopt a different strategy for the design of Code Contracts [16]. They stress that deviating from the official language should be avoided so that standard tools can be used. To that end, they propose to encode contracts using existing language features, by calling methods of a special contract library. Similarly, JML tools regularly lag behind the evolution of the language, although recent efforts have been made to find solutions for

[1]You can find this quote from George Necula in the documentation of CIL:

> When I [...] started to write CIL I thought it was going to take two weeks. Exactly a year has passed since then and I am still fixing bugs in it. This gross underestimate was due to the fact that I thought parsing and making sense of C is simple. You probably think the same. What I did not expect was how many dark corners this language has [...].

that problem [9, 23]. SPARK shares the same problem, and is currently based on a subset of Ada 2005, instead of Ada 2012.

In summary, having contracts directly in the language, and using the compiler frontend as a basis for program verification tools not only saves the initial cost of building a separate frontend, but also decreases maintenance efforts involved in keeping such a tool up-to-date with the language and its compiler.

3.2 Which Program Representation Should be Used for Formal Verification?

Using directly a compiler frontend presents some problems of its own, even though they are relatively minor compared to the maintenance issues we previously discussed. A compiler is designed to translate programs to machine code, and not to do formal verification. Typically the compiler builds and decorates various abstract representations of the program, prior to generating an executable, and the question arises as to which of these representations is best suited as input to verification tools. Taking as an example the GNAT compiler, the work that is done in the Ada frontend can be described by three phases: parsing (building the tree from the source), semantic analysis (typing, name and overload resolution, enforcement of static semantics rules), and expansion (translation of higher-level constructs). The latter two phases are actually intertwined. Expansion is a first preparation phase for code generation, breaking down high-level features of Ada into simpler constructs, or replacing them by calls to the runtime library. For example the tasking features of Ada do need a whole supporting library, Ada 2012 quantified expressions are translated into loops, and generic subprograms and packages are instantiated by some complex macro-expansion. Some normalizing transformations are also done during expansion, such as filling in default parameters and reordering the actuals for subprogram calls.

The designer of a program verification tool needs to decide where in this chain he wants to branch off from the path that leads to code generation, and create the one that leads to static verification. In Hi-Lite, we target the Why3 [5] intermediate verification language. In some sense we *do* code generation, but the target language is sufficiently different that most of the GNAT expansion phase is of no use, and may be even harmful. For example, it is preferable to keep quantified expressions in their original form, because Why3 supports quantifiers directly in assertions, and even in programs there is a better way in Why3 to encode quantified expressions than by using loops. We do not entirely disable the expansion, though, because the normalizing steps of that phase, in particular the expansion of generics, are important and would be very costly to recreate.

3.3 Target- and Compiler-dependent Proofs

Program proofs in our toolchain depend on the target and the compiler that is used, in the sense that these proofs are not valid when either is changed. This fact sets our work apart from most program verification tools, and is directly related to the fact that the formal verification tool is implemented as a backend of the compiler. At first, this looks like a disadvantage, because proofs in other systems are supposed to depend only on the language definition, if such a thing exists – proved programs can then be used in any context. In practice, however, it is difficult to pinpoint all places where the compiler is allowed to make choices or where the language behavior is undefined (see [15] for some striking examples in C), and for those places, the analysis tool can only be imprecise – because it cannot know what the compiler will choose. A typical example is the size of the machine integer types that are used for a program. To prove the absence of run-time errors (for example overflow on integer computations), a tool that is separate from the compiler will have to choose between:

- being incorrect, by assuming some fixed size which may or may not coincide with the actual compiler choice. This is of course not an option that tool builders will actually consider, but it may still occur, for example when the tool writer incorrectly uses the size of integers on the host on which the analysis is performed instead of their size on the target.
- relying on some mechanism (for example a configuration file) which contains the sizes a compiler is expected to choose for the given target platform. However, such a configuration file is rarely complete and must be kept up-to-date. As a consequence, this is a brittle approach. In this case, proofs are *not* platform-independent anymore.

By contrast, using the compiler as a base for the verification tool ensures the correctness of the tool's output because the machine integer size of the target is known to the compiler.

This consistency comes at the cost of giving up platform- and compiler-independent proofs. But in the safety-critical and embedded world, where Ada is used most, and where formal program verification is most useful, programs are generally platform-dependent, because they are mapped carefully onto the hardware, and issues of overflow are intrinsically target-dependent. Moreover, during the development of a safety-critical system, the compiler is rarely changed, as they often standardize on the behavior of a specific compiler version (with fixed compiler switches), in order to ensure binary size and run-time efficiency.

There *is* a potentially serious drawback of that approach: now *all* proofs are platform-dependent, even the proofs of components that are completely independent of such compiler choices. In particular libraries are often *designed* to be independent of the platform and compiler choices. However, if they are indeed independent, their correctness proof must be possible for any target, and establishing their correctness for a given target should amount to pushing a button, i.e. be fully automated.

One could argue that we verify neither the source code nor the object code, but some internal data structure of the compiler. While this is true, it is also true for any verification tool – it analyzes its internal tree, not the actual source file. The previous discussion has also shown that in many aspects, the tree that we analyze is closer to the actual object code than with other verification methods, while still sufficiently high-level to be amenable for formal proof.

3.4 Hybrid Verification

We have already seen in Section 2 that the existence of rich assertions in the language, and their support in the compiler, is crucial for the adoption of contract-based development. When these assertions are compiled and checked,

for example during testing, developers get immediate benefit from writing this additional code. As these assertions are just boolean expressions in Ada 2012, no new syntax has to be learned. As we already stated, executing assertions is also a good way to find bugs in the assertions themselves. Assertions are not magically correct by construction, and experience shows that errors are evenly spread between the contracts and the code.

This interaction is the first step to an integration of test and proof, which is the main objective of the Hi-Lite project. We provide a tool GNATtest to do unit testing, which benefits from but does not require contracts and so-called test-case aspects. We also provide the formal verification tool GNATprove, for which the presence of contracts is mandatory[2], and which only supports a subset of Ada 2012. However, the essence of Hi-Lite is that GNATprove tolerates subprograms that do not fall into this subset; these subprograms must then be verified by testing.

It is not obvious that such a combination of testing and proof gives meaningful results, because both verification techniques are different. Testing has the advantage that no assumptions are made, except the sometimes quite bold assumption that the testing environment is sufficiently similar to the production environment. It has the disadvantage that relatively few guarantees can be given – namely only that the program behaves as specified in a limited number of situations. On the contrary, proof guarantees the correctness of the program in *all* situations, but only when its assumptions are met. For example, GNATprove assumes for each subprogram that its arguments are not aliased nor overlapping in memory, and that all variables have values allowed by their type. GNATprove checks these assumptions on all subprograms that fall in its supported subset, but not for others. There is a potential danger that proved functions are used in a context where these assumptions do not hold.

One of the key insights of Hi-Lite is that the combination of test and proof is meaningful when the assumptions of the proof tool are checked during testing [10]. We are again at a point where having control of the compiler hugely simplifies things. In a special mode that can be activated during testing, GNAT can insert additional dynamic checks for the assumptions of formal verification. For a compiler developer, this is a small amount of work, for an outside tool this amounts to much more work [7].

3.5 Reducing the Trusted Code Base

A drawback of our approach is that we have to trust not only the deductions of the verification tool, but also large parts of the compiler. There is much research on how to reduce this *trusted code base*. A possible approach is the so-called proof-carrying code (PCC) [21], where the compiler, possibly supported by user annotations, generates object code along with artifacts that allow to independently verify that the object code has certain properties. For example, proof-carrying code can certify the absence of certain classes of runtime errors, or the absence of certain security violations. These properties can now be checked by a small checker tool, and it does not matter anymore *how* the object code has been obtained. The compiler has been removed from the trusted code base.

[2]but GNATprove will fill in default contracts when none are given.

Another approach is the *verified* compiler [19] (as opposed to the *verifying* compiler [18] approach illustrated by GNAT and GNATprove), where a compiler is implemented such that a machine-checkable proof certifies its correctness. Again, the compiler is not part of the trusted code base anymore, only the proof checker is.

We believe that the issue of the size of the trusted code-base is orthogonal to the issues described in this paper. Our compiler could very well generate proof-carrying code, and given sufficient man-power, we could very well formally prove the verification tool and the compiler, resulting in a "verified verifying compiler".

4. ENRICHING PROGRAMS AND SPECIFICATIONS

4.1 Providing Rich Libraries

It is commonly agreed that, for a programing language to be usable in practice, it must be supplied with good libraries. They allow the user to avoid rewriting standard pieces of code. They are less error prone and more efficient than what a typical user would write.

The same reasoning applies to their verification and formal proof. Libraries are more trustworthy (they have been verified) and they make the specification and the verification of programs that use them easier.

In Hi-Lite, special care has been taken to provide a library for containers well suited to proof [14]. Indeed, these data structures obviate the need for pointers, which are not easily handled in formal verification. What is more, specifications of functions on containers have been thoroughly written so that complex properties of client code can be checked.

4.2 More than Just Executable Semantics

Sometimes, executable assertions do not provide enough expressiveness, are a burden for the specification or make formal proof difficult. In such cases, it becomes necessary to enlarge the semantics of assertions.

4.2.1 Mathematical Integers in Assertions

One such case are assertions involving mathematical objects like integers. If the semantics of integer operations in assertions is the same as in the program, such operations in assertions may have to be guarded against overflow just as in program code, resulting in conditions that may be difficult to express. For example, if a programmer wants to be sure that the Ada expression `A + B` does not overflow, it is natural to write a precondition of the form `A + B in Integer`, which expresses that the result of the addition fits in the 32-bit integer type. However, the addition in this expression may itself overflow. Without giving more flexibility to the tools, the "solution" consists in writing a more complex expression with multiple cases depending on the sign of each argument, or introducing explicit conversions to a larger integer type. This burden may seem unnecessary for assertions designed to be checked by formal proof, since mathematical unbounded integers are easily handled in formal verification.

A solution to this problem is to introduce mathematical semantics for integer operations (i.e. arbitrary precision) that appear in assertions. In the absence of executable semantics, this is a natural choice, and is used for example in the B method [2]. However, it is at odds with our

goal to provide the same semantics for execution and for proof. Therefore, this solution also implies the usage of a library for unbounded integers at execution time. Such a library, on the other hand, is a constraint not every safety critical software can afford (due to the possible run-time cost, unpredictable running time, possible dynamic allocation and finally the consequences for certification of the additional library). Also, choosing different semantics for assertions and programs (whether for execution or for proof) is a potential source of confusion. For example, asserting `A + B + C in Integer` in the precondition might not prevent `A + B + C` from overflowing inside the program, depending on the concrete values of the three variables and the chosen evaluation order.

Another partial solution, which is often sufficient, consists in using a larger machine integer type for intermediate results, instead of unbounded integers. For example, all four basic arithmetic operations on 32-bit integers can be done comfortably using 64-bit integers.

Since there does not seem to be a consensus on which semantics is the best for integer operations in assertion, we have decided to provide three alternative semantics in GNAT and GNATprove. The first one is the same semantics as in programs, where assertions have to be checked against overflows in their base type. The second one uses bigger machine integer types when needed, but will fail when the larger machine integer is still not big enough. The third one is the mathematical semantics that uses an arbitrary precision library. In the second and third cases, the compiler decides, using a simple static analysis, if a larger machine integer, or even an unbounded one, is needed.

It should be noted that the Ada standard does not require the compiler to issue an overflow error when an intermediate value does not fit into the base type. As long as the final result is correct, the intermediate computations can be carried out in any possible way. Of course, the standard disallows incorrect results due to overflows. We benefit from this liberty in the standard and have implemented the two "extended" semantics described above.

We believe that both semantics are also interesting for Ada programs, not just assertions. Therefore, all three types of semantics are available for assertions and for programs, and one can choose which one to use for which. Both the compiler and the proof tools support all three types of semantics, and agree on which is used when. Of course, the implementation was simplified greatly by the sharing of a common compiler frontend. A single piece of code takes the decision on what semantics to use for each arithmetic operation, and the compiler and the verification tool then act accordingly.

4.2.2 Proof-only Theories

For user-defined packages, executable assertions may also end up being insufficient. Some properties may well be impossible to express in the executable semantics (if they use mathematical real numbers for example), or the executable specification can be too complicated for the programmer, for the solver, or for both to handle. Some non-executable constructs, such as axioms, abstract (not completely defined) functions, variables or types *etc.*, could make the specification more user-readable or more efficient for formal verification.

For example, assume we want to define a function `Count` that counts the occurrences of a given element in an array. The programmer can write an executable semantics for this function involving a recursive expression function that counts the occurrences of the element in a subrange of the array. Unfortunately, such a semantics may not allow the solver to easily deduce all needed properties, or at least not efficiently enough. Axioms or lemmas, describing the evolution of `Count` when an element is added or removed from an array for example, will make the solver's work easier.

In Hi-Lite, a Why3 [5] theory, usable only for proof, can be given for a package. This theory replaces the automatic translation of the package in the proof mechanism. Such a theory has been written for the container library for example. In future work, we plan to allow users to write these non-executable assertions directly in Ada. Indeed, that will make this mechanism usable for Ada programmers who do not know the Why3 formalism.

5. CONCLUSION

Even though many tools for formal program verification exist, very few of them have seen industrial acceptance, even when they address formal verification for an existing programming language in widespread use such as C. We believe that a necessary condition for easier adoption is the integration of the assertion language into the programming language; adoption is even smoother if both languages are essentially the same. Ada 2012 fulfills these requirements.

If this condition is met, the compiler becomes the central piece for the creation of safe software, because it understands the code and the assertion language, and can serve as a frontend for code generation, test and formal verification. This in turn makes it easier to realize and maintain tools for formal verification, that remain up-to-date in the face of advances in the language and in compiler technology. It also makes it possible to take into account platform-, compiler- and runtime-specific issues. The frontend for formal verification of the Hi-Lite project is built on top of the GNAT frontend.

Finally, we have shown that these choices are also compatible with expressing assertions in a way that is closer to a mathematical view. In Hi-Lite, we will provide a semantics of integer arithmetic for assertions that is closer to the mathematical view, and provide rich container libraries with high provability.

Acknowledgments. We would like to thank the many participants of the Hi-Lite mailing-list, in particular David Mentré, David Lesens and Stefan Lucks, as well as our colleagues and partners at AdaCore and Altran Praxis, for fruitful discussions that lead to some of the solutions presented here. We also want to thank the anonymous referees who raised some issues that are now discussed in the final version of this paper.

6. REFERENCES

[1] Eiffel : Analysis, design and programming language. Standard ECMA-367, 2d Edition (2006).

[2] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[3] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley

Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin / Heidelberg, 2005.

[5] F. Bobot, J.-C. Filliâtre, A. Paskevich, and C. Marché. Why3: Shepherd your herd of provers. In *Proceedings of the First International Workshop on Intermediate Verification Languages, Boogie*, 2011.

[6] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7:212–232, 2005.

[7] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *Proceedings of the 18th International Symposium on Formal Methods*, Paris, France, August 2012.

[8] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] D. R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 472–479, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] C. Comar, J. Kanig, and Y. Moy. Integrating formal program verification with testing. In *Proceedings of the Embedded Real Time Software and Systems conference*, ERTS^2 2012, Feb. 2012.

[11] D. Crocker and J. Carlton. Verification of C programs using automated reasoning. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '07, pages 7–14, Washington, DC, USA, 2007. IEEE Computer Society.

[12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C, A software Analysis Perspective. In *Software Engineering and Formal Methods (SEFM)*, Oct. 2012. To appear.

[13] E. Dijsktra and C. Sholten. *Predicate Calculus and Program Semantics*. Springer, New York, Berlin, 1989.

[14] C. Dross, J.-C. Filliâtre, and Y. Moy. Correct code containing containers. In *5th International Conference on Tests & Proofs (TAP'11)*, Zurich, June 2011.

[15] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM.

[16] M. Fahndrich, M. Barnett, D. Leijen, and F. Logozzo. Integrating a set of contract checking tools into Visual Studio. In *Proceedings of the 2012 Second International Workshop on Developing Tools as Plug-ins (TOPI 2012)*. IEEE, 2012.

[17] Hi-Lite: Simplifying the use of formal methods. `http://www.open-do.org/projects/hi-lite/`.

[18] C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 78–78. Springer, 2005.

[19] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[20] B. Meyer. Eiffel as a framework for verification. In *Verified Software : Theories, Tools Experiments, Forst IFIP TC2/WG2.3 Conference*, Lecture Notes in Computer Science LNCS 4171. Springer, Zurich, Switzerland, 2005.

[21] G. C. Necula. Proof-carrying code. In P. Lee, F. Henglein, and N. D. Jones, editors, *POPL*, pages 106–119. ACM Press, 1997.

[22] J. C. Reynolds. An overview of separation logic. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 460–469. Springer, 2005.

[23] Robby and P. Chalin. Preliminary design of a unified JML representation and software infrastructure. Technical report, SAnToS Laboratory, Kansas State University, 2009.

[24] C. Spear. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, New York, Berlin, 2008.