Hi-Lite - Verification by Contract

Johannes Kanig, Jérôme Guitton and Yannick Moy
AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France)
{kanig,guitton,moy}@adacore.com

Abstract

Formal methods and testing are often considered as disjoint technologies. The Hi-Lite project wants to show that both are actually complementary. The central concept are subprogram contracts, part of the upcoming Ada 2012 standard. A contract, which consists of pre- and postcondition, describes the specification of a subprogram, in the same syntax as Ada expressions. These contracts can be seen either as additional assertions in the case of testing, or they can be used to prove the correctness of the subprogram, using modern proof technology such as SMT solvers. This mechanism allows an easy adoption of modern formal methods, on a per-function basis. Hi-Lite fits in well with the upcoming DO-178C avionics safety standard, a revision to DO-178B, which, among other things, accounts for technologies such as formal methods.

A contract is additional information a programmer has to write, and errors are possible. Another focus of the Hi-Lite project is to help the programmer write meaningful and complete contracts. Current proposals include the detection of runtime errors contained in contracts, meaningless or too strong contracts, incomplete contracts that do not mention modified variables and code that does not contribute to the contract.

The goal of project Hi-Lite is to produce a verification toolchain combining formal methods and testing, integrated with the usual project structure in the two IDEs developed by AdaCore.

1 Introduction

Current practice in validation/verification of critical systems (*e.g.,* DO-178B) is centered on testing. The testing process has been prescribed to incorporate a semi-formal notion of completeness to demonstrate that the software correctly implements all its requirements, and nothing further. But in the last analysis, our confidence is based primarily on the successful completion of a test suite that, although extensive, is incomplete.

But other potentially valuable approaches are available. Static analysis, in the form of sophisticated tools that analyze application programs to detect possible problems, can help, though in only a limited fashion if the analysis is not guaranteed to be complete. A tool that promises to detect 90% of buffer overruns merely leaves us worrying even more about the 10% it has not detected.

Another important approach is the application of formal methods, which allow us to demonstrate mathematically that certain properties of programs are satisfied. However, the notion of proving an entire program correct in such a mathematical sense is a mirage. Even if we prove that a program correctly implements some complete formal specification, we have no way of ensuring that the specification itself is correct. Nevertheless, formal methods can definitely play a significant role, as is shown by the SPARK language and toolset [2] and by the use of *unit proof* at Airbus [14]. Importantly for safety- or security-critical systems, the proof tools must be sound: if they do not detect an error (such as buffer overrun), then the program does not contain any such errors.

Individually, the approaches embodied in testing, static analysis, and formal methods are useful but do not provide a complete solution. In combination, however, these techniques can complement each other and thus advance the state of the practice in developing critical software. Project Hi-Lite [7, 10] is an effort, started in May 2010 and scheduled to finish in May 2013, to build a toolset supporting such a combined approach — in particular, to ease the effort in transitioning to and adopting formal methods for software verification. This project is being conducted by a consortium of partners consisting of AdaCore, Altran Praxis, CEA LIST, Astrium Space Transportation, INRIA ProVal, and Thales Communications. Hi-Lite fits in well with the upcoming DO-178C [15] avionics safety standard, a revision to DO-178B, which, among other things, accounts for technologies such as formal methods. Static verification through formal methods may reduce the need for later testing (*e.g.,* robustness tests).

2 Mixed Static/Dynamic Verification

Previous projects [4, 3] integrating dynamic and static verification have shown the benefits of a common specification language, although the semantics attached to the specification may vary slightly depending on the techniques and tools. A central specification element is the subprogram contract, consisting in a precon-

dition and a postcondition, as defined in Design-by-Contract [11].

2.1 A Common Specification Language

The forthcoming version of the Ada standard, called Ada 2012 [13], offers a variety of new features to express properties of programs. New checks are defined as *aspects* of program entities, for which the standard defines precisely the various points at which the check is performed during execution. The most prominent of these new checks are the `Pre` and `Post` aspects which define respectively the precondition and postcondition of a subprogram. 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 (before the call) of any variable `V` as `V'Old`.

For example, a simple contract on function `Search` could specify that its parameter string should not be empty, and that the function returns the index of the first occurrence of character C in S:

```
function Search (S : String; C : Character)
   return Natural with
   Pre  => S /= "",
   Post =>
     (if Search'Result /= 0 then
          S (Search'Result) = C
       and then
         (for all X in
            S'First .. Search'Result - 1 =>
            S (X) /= C));
```

When compiled with checks on, any failure from the caller to respect the precondition of `Search` or from `Search` to respect its postcondition will be detected at run-time and reported as an error. The execution model for these aspects is simply to insert assertions at appropriate locations, which raise exceptions when violated. For each variable `V` whose pre-state value may be read in the postcondition (`V'Old`), the compiler inserts a copy of the variable's value at the beginning of the subprogram body. It is this copy which is read to give the value of `V'Old`.

Expressing properties in contracts is greatly facilitated by the use of if-expressions, case-expressions, quantified-expressions and expression-functions, all defined in Ada 2012. The main objective of these 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 1 .. 10 => P (J)` is true if-and-only-if the subprogram `P` returns `True` for every argument, starting from 1 up to 10.

The standard defines precisely the dynamic semantics of expressions occurring in contracts, which allows their implementation in a compiler for testing and run-time checking. But like any other expressions in Ada, expressions in contracts may be ambiguous (due to different orders of evaluations and compiler permissions) and contain side-effects (through writes to global variables or to memory). Neither ambiguity nor side-effects are desirable in a specification. In Hi-Lite, we restrict the language of expressions that appear in contracts to ensure that specifications have an unambiguous functional semantics. We rely on this semantics to apply static analysis tools and automatic provers to perform formal verification of programs. A subtle but crucial choice here is that we have decided to assign the same semantics to contracts in both static and dynamic settings, following the work of Patrice Chalin [5], thus avoiding subtle mistakes in specifications being overlooked during formal verification.

2.2 Test Cases as Parts of the Specification

It is a common practice to split a subprogram specification into separate parts, which are called *behaviors* in specification languages like JML, and *test-cases* in industry. To accomodate this use of test-cases as part of the specification of a subprogram, we define in Hi-Lite an aspect of subprograms called `Test_Case`, which applies to subprograms exactly like the standard `Pre` and `Post`. A test-case contains notably a `Requires` component which defines the entry condition for the test-case, similarly to a specialized precondition, and an `Ensures` component which defines the exit condition for the test-case, similarly to a specialized postcondition. An example of test-cases for an integer square-root function is:

```
function Sqrt (X : Integer) return Integer
with
   Test_Case =>
      (Name     => "normal test case",
       Mode     => Nominal,
       Requires => X < 100,
       Ensures  =>
          Sqrt'Result >= 0 and then
          Sqrt'Result < 10),
   Test_Case =>
      (Name     => "robustness test case",
       Mode     => Robustness,
       Requires => X = -1,
       Ensures  => Sqrt'Result = 0);
```

For unit testing, it is sufficient to write a test procedure which exercises a test-case to consider this test-case successful. Robustness test-cases correspond to cases beyond the normal behavior of the subprogram described in its contract, for which the subprogram contract is ignored. Other test-cases describe cases in which the subprogram contract should be respected.

For unit proof, it is sufficient to prove that the subprogram implements a special contract, with the `requires`, optionally and'ed with the original precondition, as precondition and the `ensures` as postcondition. The original precondition should be and'ed with the `requires` for those test-cases which correspond to normal behavior, and the `requires` should be the only precondition for those test-cases which correspond to abnormal behavior.

2.3 Fine-Grain Formal Verification

A common criticism of formal methods is that they cannot be applied in an opportunistic fashion to only parts of a project, where they are more cost-effective. In Hi-Lite, we can apply formal verification to individual subprograms in an otherwise unrestricted Ada program. However, some features of a complex language like Ada make formal verification orders of magnitude harder. This hardship manifests both as a high cost for specifying properties and as a lack of proof automation. Two notorious features that impede formal verification are pointers (and dynamic memory allocation/reclamation) and concurrency. We have defined a subset of Ada called ALFA which excludes such features, while being a superset of the existing SPARK subset of Ada. A common basis of SPARK and ALFA are that both define non-ambiguous subsets of Ada, in which ambiguities related to order of evaluation or compiler permissions are resolved by static restrictions on subprograms formally analyzed.

In Hi-Lite, we rely on contracts to provide the specifications against which the program should be compared to. Thus, any property that can be expressed as subprogram contracts can be verified (through testing or formal proof) in Hi-Lite. This concerns chiefly the low-level functional properties, which express the desired behaviour of subprograms, but also any other property that can be encoded in contracts, like some security properties [1] and properties that ensure absence of run-time errors. Absence of run-time errors can be classically expressed as assertions in the program, which can also be proved in Hi-Lite.

In order to prove properties (whether contracts or assertions), equivalent mathematical formulas called Verification Conditions (VCs) are generated based on properties and program code. The main bottlenecks for achieving a high level of automatic proofs are the generation of these VCs with a VC generator (VCgen) and their proof with an automatic prover. In Hi-Lite, we have chosen the recent Why VCgen [9] and Alt-Ergo [6] automatic prover. Although automatic provers are not expected to replace humans for reasoning about deeply complex properties, they can be expected to be very efficient on even moderately complex properties like the use of containers in programs [8], provided the problems are presented in a suitable form.

3 Non-Standard Verifications

Contracts have to be written by the programmer. Just as in programs, errors in contracts are possible and likely to happen. Formal methods can lure a programmer into a false sense of security, stating that the program "has been proved correct", while the contract does not express the intent of the programmer or is even meaningless. In Hi-Lite, we want to help the programmer write correct and meaningful specifications, which themselves help write correct and meaningful programs.

3.1 Absence of Run-Time Errors in the Specification

The emphasis on meaningful contracts makes even more sense in Hi-Lite than in other tools for formal methods, as contracts are not primarily logical formulas, but Ada expressions that may be executed, for example for unit testing purposes. It is not clear what should be the meaning of an assertion that, instead of returning a Boolean value, fails with a constraint error, for instance. Does that mean that the assertion is false (in the logical sense) or that it is simply meaningless?

In Hi-Lite, we adopt the latter point of view. We require that every assertion (this includes pre- and post-conditions, but also loop invariants and other assertions) *may never fail*. These additional verifications give rise to new VCs that are sent to the automatic prover for verification. In practice, this means that we do not allow, say, a precondition of the form

```
1/X > 0.
```

Instead, a programmer has to protect the division, just as he would do in program text:

```
X /= 0 and then 1/X > 0.
```

As another example, consider the following Ada code snippet:

```
function F (X : Integer) return Integer
   with Pre  => (X /= 0),
        Post => (...);

function G (X : Integer) return Integer
   with Pre  => (F(X) > 0),
        Post => (...);
```

Again, this would not be accepted as is by the Hi-Lite tools, because the precondition of `G` may fail when `X` is equal to zero. As before, the programmer must protect the call to `F` in the precondition of `G`, as follows:

```
function G (X : Integer) return Integer
   with Pre  => (X /= 0 and then F(X) > 0),
        Post => (...);
```

The aim of these restrictions is that each contract should be *complete* in the sense that all requirements of the contract are immediately visible and do not depend on the contracts of other subprograms.

3.2 Absence of Unintended Functionality

Another concern is that contracts should entirely summarize the purpose of a subprogram. If they don't, they are probably wrong or at least incomplete. To illustrate the issue, consider the following procedure sketch:

```
procedure P (X : Integer)
   with Pre  => (...),
        Post => (if X = 0 then ...);

procedure P (X : Integer) is
begin
   if X = 0 then
      --  Do something when X = 0
   else
      --  Do something else otherwise
   end if;
end P;
```

Here, the problem is that the contract only states the behavior of the procedure when `X` is equal to zero, but not what happens when this is not the case. This means that the entire `else` branch does not contribute to establishing the postcondition. This introduces a semantic notion of "dead code": the code in the `else` branch is "dead" in the sense that outside the procedure `P`, no other part of the code should take advantage of the effects in that branch. We believe that it would be of great help for the programmer to report this situation, indicating which portion of the code is "dead" in this sense. The programmer can now either correct the contract to reflect both situations or remove the offending portion of the code. This detection will rely on the automatic prover ability to record the set of hypotheses it used to prove a VC. Such a feature is currently developed for the Alt-Ergo prover.

Another case of incomplete specifications is illustrated by the following simple program:

```
procedure Full_Stop
   with Pre  => (...),
        Post => (Accel = 0);
procedure Full_Stop is
begin
   Accel  := 0;
   Breaks := On;
end Full_Stop;
```

In this example, the contract is again incomplete: it only mentions that the acceleration is set to zero, but not that the breaks are activated. Said otherwise, it only mentions the modification of the `Accel` variable, but not the one of `Breaks`. Again, a warning could be issued to the programmer, stating that a written variable is not mentioned in the contract, so no other part of the program can be aware of its new value, and this is probably a bug either in the code or in the contract. In this particular example, the previously mentioned warning about code that does not contribute to the postcondition would be issued as well, but other situations, that would only be detected by the analysis concerning effects, are possible.

3.3 Redundant Specifications

A common case of meaningless specifications is the case of trivial or redundant assertions. An assertion that is always false or always true is not very useful. Worse, a *precondition* that is always false (or *inconsistent*) makes the corresponding subprogram trivially *correct*, because under this false hypothesis, everything can be proved. Similarly, a postcondition that is always true can be proved correct, but it certainly does not express anything interesting about the subprogram. Moy and Wallenburg [12] detected cases of such irrelevant annotations in the code of Tokeneer project [1] which had been formally proved in SPARK.

In Hi-Lite, we propose to detect such undesirable annotations and to issue a warning to the programmer. In practice, detecting an inconsistent precondition amounts to trying to proving `False` just after assuming the precondition. If the proof succeeds, anything can be proved at that place in the code, so the precondition must be inconsistent. A trivial postcondition that is always true can be detected by trying to prove it in the *empty context*, that is, without assuming the precondition to be true nor the subprogram body to execute correctly.

4 Addressing Shortcomings of Formal Methods

There are numerous concerns that have been raised against formal methods. In this section, we argue that the part of Project Hi-Lite concerning proofs of programs addresses some of them, while testing the program takes care of most of the others.

Dewar [7] lists a number of potential problems with the application of formal methods:

- Formal specifications with logical formulas are hard to write and hard to read, and must in consequence be written by specialists. They can also become quite large, as large as the program to be considered or even larger.

- Specifications, as programs, can contain errors. Contrary to programs, specifications can usually not be tested, as they are logical formulas. Finding errors in the specifications is difficult.

- Formal specifications tend to omit low-level details that may nevertheless be relevant to the overall correctness of the program.

- Usually, one only proves *partial* correctness, *i.e.,* the fact that the program is correct *if it terminates*. Termination itself is often not proved, but of course a (correct) subprogram that never terminates can be as problematic as a run-time error.

- In most practical cases (consider C, C++, Ada or Java), the semantics of the programming language to which formal methods are applied is ill-defined. A compiler may, inadvertently or deliberately, apply optimizations that are not considered by the formal method in question.

Project Hi-Lite uses a single language for programs, test cases and formal specifications. This makes contracts not harder to write than the program itself and eliminates the first concern. In Section 3, we have made a number of concrete proposals to detect errors in specifications and to help the programmer write correct and meaningful contracts. The remaining problems are exactly the ones that are dealt with by tests. So the dual nature of Hi-Lite, using tests and formal methods in a complementary way, gives the programmer a means, not to eliminate these remaining problems, but at least to detect them.

5 Conclusion

The upcoming standard DO-178C accounts for the usage of formal methods to ensure the conformity of a program with respect to its specification. Many formal methods are only meaningful when applied to an entire program, and thus are extremely costly to apply to existing projects. Also, without considerable effort, formal methods cannot entirely *replace* testing. Project Hi-Lite, on the contrary, allows a smooth migration from unit testing to unit proof, on a per-subprogram basis. Testing and proving become complementary activities. In addition, the project Hi-Lite proposes innovative forms of help for the programmer to write meaningful contracts.

References

[1] Janet Barnes, Rod Chapman, Randy Johnson, James Widmaier, David Cooper, and Bill Everett. Engineering the Tokeneer enclave protection software. In *1st IEEE International Symposium on Secure Software Engineering*, March 2006.

[2] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec♯ programming system: An overview. In *In CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[4] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7:212–232, June 2005.

[5] Patrice Chalin. Engineering a sound assertion semantics for the verifying compiler. *IEEE Trans. Softw. Eng.*, 36:275–287, March 2010.

[6] Sylvain Conchon and Evelyne Contejean. The Alt-Ergo automatic theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[7] Robert Dewar. A pragmatic view of formal methods: the Hi-Lite project. In *Advances in Systems Safety, Proceedings of the 19th Safety-Critical Systems Symposium*, pages 521–524. Springer, February 2011.

[8] Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy. Correct code containing containers. *Test and Proof*, 2011. `http://www.open-do.org/wp-content/uploads/2011/01/main.long.pdf`.

[9] Jean-Christophe Filliâtre and Claude 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.

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

[11] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[12] Yannick Moy and Angela Wallenburg. Tokeneer: Beyond formal program verification. *Presented at Embedded Real Time Software and Systems*, May 2010.

[13] Edmond Schonberg. Towards Ada 2012: an interim report. In *Proceedings of the ACM SIGAda annual international conference on SIGAda*, SIGAda '10, pages 63–70, New York, NY, USA, 2010. ACM.

[14] Jean Souyris, Virginie Wiels, David Delmas, and Hervé 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.

[15] RTCA SC-205/EUROCAE WG-71. DO-178C. software considerations in airborne systems and equipment certification. `http://ultra.pr.erau.edu/SCAS/`.