# Integrating Formal Program Verification with Testing

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

**Abstract**

Verification activities mandated for critical software are essential to achieve the required level of confidence expected in life-critical or business-critical software. They are becoming increasingly costly as, over time, they require the development and maintenance of a large body of functional and robustness tests on larger and more complex applications. Formal program verification offers a way to reduce these costs while providing stronger guarantees than testing. Addressing verification activities with formal verification is supported by upcoming standards such as DO-178C for software development in avionics. In the Hi-Lite project, we pursue the integration of formal verification with testing for projects developed in C or Ada. In this paper, we discuss the conditions under which this integration is at least as strong as testing alone. We describe associated costs and benefits, using a simple banking database application as a case study.

**Keywords:** Formal verification, Testing, Verification by contract, DO-178, Ada 2012

## 1 Introduction

In critical software development, such as avionics software, program verification techniques are heavily based on the notion of requirement based testing (RBT). The DO-178B[1] [16] standard for avionics software development defines several levels for these requirements. Some of the system requirements are allocated to software and are the basis for high level software requirements (HLRs). From those HLRs, a software architecture is designed that is made concrete through low level requirements (LLRs). The latter are traditionally expressed in natural language and attached to subprograms. RBT consists in defining test cases associated with a given requirement, verifying that those test cases are sufficient to cover all aspects of the informal requirement and then create test procedures that implement the test cases. DO-178 (6.4) defines 3 levels of testing: hardware/software integration testing (to be done on the final hardware), software integration testing and low level testing, together with testing objectives that ensure an adequate coverage of the program behaviors despite the non-exhaustive nature of testing. These testing objectives, such as Modified Condition/Decision Coverage (MC/DC) [10], account for a large part of the high cost of testing, together with robustness testing. The goal of robustness testing is to assess the behavior of the software in unexpected situations, such as error cases. It is therefore of practical value to turn to alternate verification techniques which can provide at least as strong results at a lower cost.

Formal methods have the potential for analyzing exhaustively all behaviors of a program, thus reaching 100% coverage. Contract-based formal program verification, which consists in verifying statically that a subprogram respects its contract, has already been pioneered in the avionics industry on C and SPARK programs [18, 12]. The upcoming DO-178C Formal Methods Supplement recognizes such usage and explicitly gives permission to replace some of the prescribed testing activities by formal methods. In this paper, we describe how low level testing can be be *partly* replaced by contract-based formal verification. We do not aim at completely replacing low level testing with formal verification, because the latter may be much too difficult (hence, costly) for some parts of the program. Instead, we aim at fully automated formal verification on parts of the program, completed with low level testing on the parts that are not formally verified. The essential difficulty here is to ensure that the combination is as strong as testing alone.

[1] The B version of DO-178 currently in use will be replaced in 2012 by the C version. In this paper, we denote DO-178 both these versions.

## 1.1 Formal Executable Contracts

For formal verification, LLRs must be expressed in a formal language. In contract-based formal verification, LLRs are expressed as subprogram contracts. A contract is given by a precondition and a postcondition as in Hoare logic [14], expressed in a specification language [13]. The precondition denotes a property that a caller must guarantee before calling the subprogram. The postcondition denotes a property that the subprogram must guarantee before returning to the caller. Contract-based formal verification consists in proving separately that each subprogram respects its contract and does not contain possible run-time errors (*e.g.,* division by zero), and that the preconditions of subprograms it calls hold. It is worth noting that during the verification of a given subprogram, assumptions are made that need to be verified at some point:

- the subprogram's precondition holds at subprogram entry;
- after a call, the callee's postcondition holds.

In some specification languages, like SPARK annotations [4], contracts are expressed in a logic formalism different from the programming language, while others, like JML [8], mix programming language constructs and logic ones. We have chosen instead a fully executable specification language, integrated in the programming language, as in Design-by-Contract [15]. Since contracts are written by humans, they are as likely to contain errors as programs. By choosing contracts that are also code, they can be easily debugged and tested, using the usual development tools. In particular, run-time errors could be raised when executing contracts, which should be detected by formal verification, like run-time errors in code. Thus, contracts can be both interpreted as logic formulas and as *pure* code (whose execution cannot modify the global state). Additionally, experiments by Chalin [9] have shown that having the same semantics for code and contracts favors adoption by non-expert developers.

## 1.2 Implicit Contracts

The main cost incurred in contract-based formal verification is the cost of writing subprogram contracts. It is kept to a minimum by a series of design choices.

First, in strongly typed languages, the types of variables and subprogram parameters already define a rich default contract. To leverage on this default contract in contract-based formal verification, it is necessary to exclude language constructs which circumvent the type system. Moreover, an implicit precondition is needed to guarantee that input subprogram parameters have values allowed by their type. Similarly, an implicit postcondition guarantees that output subprogram parameters (and function result) have values allowed by their type.

Second, contract-based formal verification requires knowing the subprogram data dependencies, that is to say which global variables are read and/or written by a subprogram, directly or indirectly. Asking the programmer to provide those data dependence's manually considerably increases the cost of writing contracts. Computing those automatically is therefore a natural choice. As consequences of this choice, the complete program must respect some restrictions, and a subprogram can be formally verified only if all the subprograms it calls, directly or indirectly, have already been implemented. This constraint is similar to the one for testing a subprogram.

Last, the complexity of contract-based formal verification is greatly increased in the presence of object aliasing: the manual effort to write contracts increases, and the number of automatic proofs decreases. The most effective choice is to restrict the use of pointers in the analyzed language subset, so that there is no aliasing between parameters themselves, and between parameters and global variables accessed directly.

On one side, implicit preconditions and postconditions are assumed during the verification of a subprogram: its implicit precondition is assumed at subprogram entry, and implicit postconditions of subprograms called are assumed. On the other side, implicit preconditions and postconditions are additional verification goals during the verification of a subprogram: its implicit postcondition should be proved, and implicit preconditions of subprograms called should be proved.

## 1.3 Combining Testing and Formal Verification

While formal verification gives stronger guarantees than testing, these guarantees depend upon the assumptions described in Sections 1.1 and 1.2. In a context where all subprograms are formally verified, these assumptions are met automatically. They are either discharged by meta-level reasoning, static analysis or proof. One of the keys to combining testing and formal verification is the ability to verify these assumptions on the parts of the code that are not formally verified. Whether the combination happens at the subprogram level or at a larger software component level, the issues are the same: assumptions on properties of other parts of the code are made, and those assumptions must be verified.

There are several ways of combining testing and formal verification. The simpler case (denoted here T+F.1) is to divide the application into chunks (*e.g.,* subprograms) and apply completely one of the verification methods for each chunk. Another possibility (denoted T+F.2) is to verify each separate LLR with a given verification method. Thus, when a subprogram implements several LLRs, it should be possible to apply both techniques for partial verification of the same code. A third possibility (denoted T+F.3) is to demonstrate specific formal properties that would replace part or all of robustness testing.

## 1.4 A Case Study

We are currently developing two toolchains for combining testing and formal verification of programs in project Hi-Lite [2]. AdaCore is developing a toolchain for projects in Ada, based on the Ada GNAT compiler in GCC. CEA is developing a toolchain for projects in C, based on the Frama-C platform and GCC compiler. Both platforms share a common proof back-end developed by INRIA, based on the Why3 [7] intermediate language.

GNATprove [11] is our tool for contract-based formal verification of Ada subprograms. GNATprove defines a complete proof compilation chain, from source programs in a large subset of Ada to Why3 [7], from which it generates Verification Conditions (VCs) that are proved with the Alt-Ergo automatic prover [6]. A VC is a logical formula whose validity must be proved to ensure that a corresponding assertion in the source Ada program is verified. Note that GNATprove does not verify that a subprogram terminates.

In this paper, we describe the approach chosen in GNATprove. We present examples in the context of a small case study in Ada, consisting in a simple banking database application. HLRs for this application are the following:

1. It should be possible to open and close empty accounts attached to named individuals uniquely determined by an identifier (*e.g.,* SSN, ID), to deposit and withdraw money on an account, and to query the account balance.
2. There is a maximal number of 200 000 accounts that can be in use simultaneously.
3. An account is in a fixed currency decided at opening. All deposits and withdrawals on this account should be in the same currency.
4. Accounts cannot hold more than 1 million, whatever the currency.
5. The bank does not lend money. The balance should always be non-negative.

LLRs are derived from HLRs by defining a base API for package `Database` consisting in five subprograms, together with a specification of their functionality:

- `Balance`: query the current balance of an existing account.
- `Open`: open a new account. The maximal number of accounts should not be reached already. The newly created account is attached to the given name and identifier, and it has a balance of zero.
- `Close`: close an existing account attached to the given name and identifier. The balance should be zero prior to closing the account.
- `Deposit`: deposit an amount of money on an account. The amount should be in the same currency as the account, and after deposit the balance should not exceed 1 million.

- `Withdraw`: withdraw an amount of money from an account. The amount should be in the same currency as the account, and less than the current balance.

The database is implemented in a package `Database`, which depends on two packages `Identity`, defining the types of names and unique identifiers, and `Money`, defining the types of currencies and amounts, with basic arithmetic operations over amounts. The complete code for this application (<1kloc) is available online [1]. It might be necessary to browse the code in order to understand the details of the examples provided to illustrate this paper.

# 2 Formal Verification

Formal verification of subprogram contracts and verification of absence of run-time errors both consist in proving that a set of assertions on the source program hold in every possible execution. This is only possible if the source code is restricted in some ways, and if subprogram contracts can be expressed as assertions.

## 2.1 Language Restrictions

The source language we consider is Ada 2012 [17], the forthcoming version of the Ada standard, currently supported by the GNAT compiler. Ada 2012 is designed to be as much upward compatible as possible with older versions of the language, so any legacy Ada code can benefit from this approach.

We call *Alfa* [3] a very large subset of Ada 2012 suitable for formal verification. Formal verification is only available for those subprograms whose implementation is in Alfa. We call *Alfa Friendly* [3] an even larger subset of Ada 2012 that doesn't necessarily have the right characteristics for formal verification, but can be mixed safely with Alfa code. For example, a subprogram in Alfa can call a subprogram not in Alfa (as long as it is Alfa Friendly) and the other way around. Restrictions of Alfa Friendly code ensure that some global assumptions can be made during formal verification of Alfa code. Let's now give an idea of the restrictions associated with Alfa.

First, Alfa restricts language features to remove potential ambiguities, because otherwise no formal verification is possible. Most notably, it does not allow functions that have side-effects by writing global variables (this does not apply to procedures). Since the only expressions with such side-effects in Ada are function calls, Alfa expressions cannot write global variables, which prevents any compiler-dependent behavior due to differences in the order of evaluation of expressions.

Second, Alfa restricts language features to make automatic proof possible. Indeed, some language features make automatic verification notably harder or even intractable. Since we aim at entirely automatic verification, Alfa excludes those problematic language features: exceptions, pointers (*access types* in Ada) and tasking. Note that Alfa allows passing parameters by reference in subprogram calls, as done in SPARK, which the compiler translates as pointers in the executable. In the future, limited versions of those excluded features could be considered for inclusion in Alfa.

## 2.2 Formal Contracts

Ada 2012 offers a variety of new features to express properties of programs. The most prominent of these new features 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 (at the beginning of the call) of any variable or parameter `V` as `V'Old`. Expressing properties in contracts is greatly facilitated by the use of new Ada 2012 expressions: conditional expressions, case distinction expressions, universally and existentially quantified expressions, and functions whose body is defined by a single expression (also known as expression functions). Because Alfa expressions cannot have side-effects, and because we ensure there cannot be run-time errors in contracts by formal verification, formal contracts also have an interpretation as logic formulas.

In our example, in order to express LLRs as contracts in Alfa, we first need to define helper functions which query the state of the database:

- `Existing`: says if an input account exists.

- `Belongs_To`: says if an input account is attached to a given name and identifier.
- `Max_Account_Reached`: says if the maximal number of accounts has been reached.
- `Currency`: gives the currency of an input account.

These small query functions are best expressed as expression functions, which may directly inspect some global data holding the state of the database, or call other functions to do so. For example, function `Existing` is defined as the following expression, containing itself a call:

```
function Existing (Account : in Account_Num) return Boolean is
   (not Is_Available (Account));
```

Since the Alfa function `Is_Available` does not have side-effects, the following postcondition is implicit and automatically satisfied:

```
function Existing (Account : in Account_Num) return Boolean
with
  Post => Existing'Result = not Is_Available (Account);
```

Using these functions, all the LLRs expressed in Section 1.4 can be expressed as Alfa contracts. As an example, here are the contracts of `Balance` and `Open`:

```
function Balance (Account : in Account_Num) return Money.Amount
with
  Pre => Existing (Account);

procedure Open
  (Customer : in      Identity.Name;
   Id       : in      Identity.Id;
   Cur      : in      Money.CUR;
   Account  :    out Account_Num)
with
  Pre  => not Max_Account_Reached,
  Post => Existing (Account)
            and then Belongs_To (Account, Customer, Id)
            and then Money.Is_Empty (Balance (Account));
```

## 2.3 Verification of Implicit Contracts

As described in Section 1.2, implicit contracts for data dependencies should be correct by construction. Implicit contracts for strong typing and non-aliasing, on the contrary, should be explicitly verified.

For strong typing, it is sufficient to generate VCs for the corresponding implicit preconditions and postconditions, which can be proved like other VCs. In our example, on procedure `Open`, we need to generate three VCs to show that input parameters `Customer`, `Id` and `Cur` have values allowed by their type when calling `Open`; and we need to generate one VC to show that output parameter `Account` has a value allowed by its type when returning from `Open`.

Note that we do not assume that global variables read and/or written have values allowed by their type, unless they are initialized at declaration. If needed, the user should add the corresponding preconditions and postconditions. In the case of procedure `Open`, all four global variables read and written in `Open` are initialized at declaration.

For non-aliasing, it is sufficient to check statically at each call that no pair of subprogram arguments, or a subprogram argument and a global variable read or written (as given by the computed data dependencies), are aliased, unless:

- both are only read (*e.g.,* two input-only parameters), because their aliasing is not a problem for formal verification, or
- one of them is an input-only parameter of scalar type, because it is passed by value (never by reference), so there is no possible aliasing.

For example, procedure `Open` reads and writes four global variables: scalar `First_Available` and arrays `Links`, `Accounts` and `Accounts_Balance`; and it has one input-only parameter of scalar type `Cur`. Thus, for each call to `Open`, we must check that the following pairs of arguments and global variables are not aliased: `Customer` and `Account`, `Id` and `Account`, and any pair of an argument in (`Customer`, `Id`, `Account`) and one of the global variables. Most of these pairs cannot be aliased due to the strong typing guarantees of Alfa. Therefore, we only need to check that the argument `Account` and the global variable `First_Available` are not aliased. Since the only possibility of aliasing in Alfa is through named references, this amounts to a simple semantic check, like done in SPARK or Why3.

## 2.4 Verification of User-Defined Contracts

For user-defined contracts, a VC should be generated for each assertion that should hold for the source program, and given as input to an automatic prover.

In our example, in the case of function `Balance`, there is no postcondition to prove. However, function `Balance` has a precondition, which should be proved at each call to `Balance`. This is the case both in code and in contracts, since even in contracts a function call must respect its precondition. For example, a VC should be generated to show that the precondition of `Balance` is satisfied when called in the postcondition of `Open`. Here, this can be proved easily because the required property `Existing (Account)` is already in the postcondition of `Open`.

In the case of procedure `Open`, there is a postcondition to prove. So a VC should be generated to show that, in any calling context satisfying the precondition of `Open`, its implementation returns in a state where the postcondition holds. Both the precondition and postcondition of `Open` contain calls to functions: `Max_Account_Reached` in the precondition, `Existing`, `Belongs_To` and `Money.Is_Empty` in the postcondition. The way calls are handled in proofs is to abstract each call by the underlying function contract. Here, all functions used in contracts are defined as expression functions, which can be abstracted by their implicit postcondition.

GNATprove proves automatically all contracts in our example: 17 preconditions on calls and 7 postconditions on subprograms.

## 2.5 Verification of Absence of Run-Time Errors

For each possible run-time error that can be expressed as an assertion (*i.e.,* all run-time errors in Alfa code except stack overflow), a VC should be generated and given to an automatic prover.

Like code, contracts can raise run-time errors since they are executable. Hence, VCs should be generated and proved for possible run-time errors in contracts. Preconditions deserve a special treatment as it is not known in general in which context the subprogram may be called. In order to completely rule out run-time errors in preconditions too, preconditions should be self-guarded, so that no run-time errors can be raised in any context.

In our example, the implementations of function `Balance` and procedure `Open` may raise exceptions for range checks on scalar subtypes and index checks on array accesses. In both subprograms, the precondition and the postcondition cannot raise a run-time error (or only inside a call, but the corresponding functions are also formally verified.)

GNATprove proves automatically all run-time checks in our example: 32 index checks, 16 range checks and 10 overflow checks.

## 2.6 Consistency Checks

Like code, contracts are written by humans and thus can contain errors. We have devised various ways in which we could detect inconsistencies in contracts, in order to catch errors as early as possible, even before an implementation is available. In each case, it consists in generating specific VCs for additional checks: redundant annotations, inconsistent annotations and unimplementable contracts.

- A redundant annotation is the repetition of a known fact, which may not be obvious to the user. Yet, it can be shown also in a simple example:

```
Pre => Existing (Account) and then not Is_Available (Account);
```

Here, since `Existing` is defined as the opposite of `Is_Available`, the precondition above is redundant.

- An inconsistent annotation is an annotation that can never be satisfied, which again may not be obvious to the user. Using the same example as above, a simple example of inconsistency is:

```
Pre => Existing (Account) and then Is_Available (Account);
```

- An unimplementable contract has the following characteristic: there are values of variables on entry to the subprogram, such that the precondition holds, for which no values of variables on exit from the subprogram satisfy the postcondition. In other words, no implementation can respect the contract. It is not always obvious how to detect unimplementable contracts by review as shown by the following contract:

```
function Sign (X : Integer) return Integer with
   Post => (if X <= 0 then Sign'Result = -1)
           and then (if X >= 0 then Sign'Result = 1);
```

Here, there is a problem for X = 0, because it will be impossible to find a result that is both equal to -1 and 1 at the same time.

Note that the VCs generated for consistency checks are not related to the verification of contracts or absence of run-time errors. We are only interested in the possibility of proving (or disproving) some of them, not all of them. The VCs proved (or disproved) will pinpoint redundant, or inconsistent, or unimplementable contracts, but we are not aiming at proving that a contract is not redundant, or consistent, or implementable.

# 3 Mixed Verification

We wish to apply formal verification as broadly as possible, to Ada programs containing subprograms in Alfa and subprograms not in Alfa (because they are using pointers, exceptions, *etc.*). We are interested in a methodology that allows applying formal verification to subprograms in Alfa, and testing to subprograms not in Alfa. Clearly, the results of the combination cannot be as strong as those provided by complete formal verification. Instead, we aim at a combination whose results are at least as strong as those of testing alone. This allows replacement of testing by formal verification for subprograms in Alfa.

## 3.1 Language Restrictions

We must restrict the language used for the complete program, in order to be able to discharge on the testing side the assumptions made by formal verification. Still, we want to allow exceptions and pointers, that are not allowed in Alfa.

Data dependencies are transitive. In order to compute accurately data dependencies of Alfa subprograms subject to formal verification, we need to compute them as well on all Alfa Friendly subprograms that are called from Alfa subprograms. In order to guarantee the correctness of computed data dependencies, we distinguish between those global variables that cannot be pointed to, and those global variables that can. The former are the only global variables that are allowed in Alfa. The latter are collectively referred to as a single logic variable `Heap`, which represents all locations reachable indirectly. We also forbid calls through subprogram pointers, whose data dependencies are not easily computable, and controlled types, whose finalization requires the insertion of calls at many different places in the call-graph, thus complicating the computation of data dependencies.

The restrictions above are enough to guarantee that aliasing can be detected when it might invalidate assumptions made by formal proofs. In particular, by forbidding calls through subprogram pointers, we know statically for each call which subprogram is called, and its data dependencies. This allows performing a global static analysis to detect aliasing between call arguments and global variables in data dependencies.

The above restrictions define the *Alfa Friendly* subset of Ada 2012. They guarantee that implicit contracts for data dependencies, strong typing and non-aliasing can be verified. Formal definitions of

the Alfa and Alfa Friendly subsets of Ada 2012 are part of the Hi-Lite project deliverables [3]. Some of these restrictions could be lifted later on if needed, for example calls through subprogram pointers could be allowed by considering all possible subprogram candidates. To summarize, in our context, mixing formal verification and testing can only be achieved with programs that are entirely in the Alfa Friendly subset, while formal verification applies to Alfa subprograms within an Alfa Friendly application.

### 3.2 Verification of User-Defined Contracts

The first case to consider is when a tested subprogram `T` calls a proved subprogram `P`. The verification of `P` relies on the assumption that its precondition is respected at each call site. This assumption can be discharged when testing `T` by executing the assertion that `P`'s precondition holds.

The second case to consider is when a proved subprogram `P` calls a tested subprogram `T`. The verification of `P` relies on the assumption that, whenever `T` is called in a context that respects its precondition, then `T` returns in a state where its postcondition holds. First, the precondition of `T` should reflect its LLRs, so that it is correct to call `T` in all contexts where its precondition holds. Second, the assumption above should be checked when testing `T` by executing the assertion that its postcondition holds. Although the default postcondition of "true" always holds, it might be necessary to write a more precise postcondition for `T` reflecting its LLRs, in order to formally verify `P`.

Both verifications are only possible because contracts are executable. It is still to be investigated how much testing based on those executable contracts is needed in a DO-178C context, when part of the LLRs are verified formally. One might consider running all the software integration tests in such a mode for that purpose. Such testing with all contracts verified at run time can also be used as an argument of property preservation between the source code and the object code for those properties that have been verified formally at the source level.

### 3.3 Verification of Implicit Contracts

Implicit contracts for data dependencies should be correct by construction. The verification of assumptions related to strong typing and non-aliasing requires that additional checks are performed at run-time.

Implicit contracts for strong typing can always be verified by testing in the same way as for user-defined contracts, by executing the corresponding assertions. In our example, on procedure `Open`: three assertions should be executed at subprogram entry, checking that input parameters `Customer`, `Id` and `Cur` have values allowed by their type; one assertion should be executed at subprogram exit, checking that output parameter `Account` has a value allowed by its type.

```
procedure Open
  (Customer : in      Identity.Name;
   Id       : in      Identity.Id;
   Cur      : in      Money.CUR;
   Account  :    out Account_Num);
```

Note that this is not the same as detecting lack of initialization. If a variable is initialized, then its value is allowed by its type, but the opposite is not always true: for example, a variable of machine integer type always has a value allowed, even when it is not initialized.

For non-aliasing between two subprogram arguments, we should check at subprogram entry that the objects denoted by the pairs of arguments described in Section 2.3 do not overlap. For procedure `Open`, a check is needed whenever `Customer` and `Account` might overlap, or `Id` and `Account` might overlap, for example when passing in arguments locations pointed to.

For non-aliasing between a subprogram argument and a global variable read or written, a global static analysis is needed. Suppose a proved subprogram `P` can be called in a context where such aliasing occurs between one of its arguments and a global variable `G` it reads or writes. We assume this is not one of the cases allowed in Section 2.3, in which both are only read, or one is an input-only parameter of scalar type. `G` is either an Alfa global variables, whose address cannot be taken, or the special logic variable `Heap`. For the aliasing to occur in the first case, `G` must have been passed explicitly in argument to a function call, either to `P` or one of its callers `Q`. Since `G` is read or written by `P`, it is also read or written by `Q`. Since it is passed in argument to `Q`, it is a global variable for `Q`. By statically rejecting such calls to a subprogram `Q` where an argument is aliased with a global variable read or written, we avoid such

aliasing. For the aliasing to occur in the second case, similarly, a part of `Heap` must have been passed explicitly in an argument to a function call which has `Heap` in its data dependencies. We also statically reject such calls. For procedure `Open`, this amounts to rejecting all calls which pass the global variable `First_Available` as argument `Account`.

In practice, we will define a special mode of the GNAT compiler, so that additional assertions are inserted in the executable to perform the run-time checks described above.

### 3.4 Formal Test Cases

In general, testing a subprogram is performed with manually coded oracles which translate the LLRs for the subprogram into code. If these oracles are coded as assertions, then they can also be coded as a subprogram contract. Then, testing the subprogram reduces to calling it on suitable inputs, since compiling the code in the appropriate mode already inserts assertions for contracts in the executable.

A contract describes the overall expected behavior of a subprogram in an expected context. When verification is based on testing, we recommend exercising the subprogram on representative inputs that "cover" the input space. In other words, there must be enough test cases in order to functionally cover the LLR. Furthermore, since testing is non exhaustive, it may be the case that the subprogram can be called in an unexpected context, for which a desired outcome should be specified in order to make the code robust to such unexpected situations. DO-178 differentiates these two kinds of test cases:

- the *normal range* test cases, that verify the subprogram with respect to its requirement, and
- the *robustness* test cases, that verify the behavior of the subprogram in abnormal circumstances.

In many cases, these test cases can be expressed as additional contracts that a subprogram should respect. Using the permission in Ada 2012 to create compiler-dependent aspects, we have defined in the GNAT compiler a new `Test_Case` aspect that allows users to describe test cases in subprogram specifications, along with contracts. A test case is either in nominal mode (to test in an expected calling context) or robustness mode (to test in an unexpected calling context). A nominal test case may refine the subprogram contract by defining a requires-clause (added to precondition) and an ensures-clause (added to postcondition). A robustness test case replaces the subprogram contract with its requires-clause and ensures-clause so that the program can be exercised outside of the range of its contract.

For example, two test cases can be defined for function `Balance` as follows:

```
function Balance (Account : in Account_Num) return Money.Amount
with
  Pre => Existing (Account),
  Test_Case => (Name     => "existing account",
                Mode     => Nominal,
                Ensures =>
                  Balance'Result.Currency = Currency (Account)),
  Test_Case => (Name     => "unknown account",
                Mode     => Robustness,
                Requires => not Existing (Account),
                Ensures  => Balance'Result.Currency = None);
```

The test case "existing account" exercises the nominal mode of function `Balance`, in which its contract should be respected. Here, we want to test additionally that the currency of the result is the same as the currency of the account. The test case "unknown account" exercises function `Balance` outside its nominal mode. Here, we ignore the contract of `Balance`. Instead, we require that the `Balance` be called in a context where the input account does not exist, and that the result is in a default invalid currency.

The definition of formal test cases allows automating the verification of functional coverage of the subprogram contract (*i.e.,* its requirement) by its test cases. We have developed a tool called GNATtest for unit testing, based on the xUnit family of testing frameworks [5]. It supports Ada subprograms specified with formal contracts and formal test cases. GNATtest generates a test harness for a project, with a separate test procedure for each test case, or a single test procedure for a subprogram when no test case is defined. The user only has to fill each of these procedures with a proper call sequence to to the subprogram under test, then simply compile and run the test harness, to get a report detailing which test cases and contracts are adequately tested.

# 4 Conclusion

Advances in state-of-the-art automatic provers make it possible to get strong static guarantees about programs in a cost-effective way. In order to integrate such formal verification activities into the usual verification processes, it is necessary to combine their results with those of testing. This is what project Hi-Lite achieves, as exemplified throughout the paper on a simple banking database application. The main cost incurred to apply contract-based formal verification is to formalize the properties of interest as contracts at the subprogram level, which is also a known benefit by itself: formal specifications lead to fewer defects even when no analysis is applied. The main benefits are the many different analyses that can be performed from contracts: testing, proof of absence of run-time errors, proof of contracts, and rich consistency checks on contracts even before the code is developed.

This paper presented the combination of results for subprograms that are either completely formally verified or completely tested, which we called T+F.1 in introduction. We showed that this combination is as strong as testing alone, provided that the code of the complete program follows some restrictions, and that additional run-time checks are executed during testing. Industrial case studies based on the tools described in this paper have started, and are expected to be completed by 2013.

We foresee an interest in combining formal verification and testing at a finer-grain level. First, we could use formal verification to prove that formal test cases completely cover all aspects of the requirement expressed as a contract. As an example, consider the two test cases of the subprogram `Open` in our database example:

```
Test_Case => (...
              Requires => Num_Accounts = 0,
              ...),
Test_Case => (...
              Requires => Num_Accounts > 0,
              ...),
```

Here we clearly see that the two test cases partition the input space of `Open` in two disjoint spaces, and cover it completely, as `Num_Accounts` is defined as a natural number. This can be expressed as a logic formula: the `Requires` of all test cases, or'ed together, should be equivalent to the precondition.

Second, test cases may be useful in partitioning the formal verification of a subprogram. If the complete formal verification of subprogram `Open` turns out to be too difficult, we could at least try to prove the contract in *some of the test cases*, instead of the general case. This combination of formal verification and testing at the test case level, which we called T+F.2 in introduction, would require a new way to show some level of completeness of the requirement based verification. The completeness provided by coverage analysis, which is required when testing is used alone, is not applicable to a combination of testing and formal verification on the same subprogram.

Third, contracts can in some cases be completed to reflect the expected behavior in abnormal contexts. By proving these contracts, one fulfills the objectives of robustness testing, which we called T+F.3 in introduction. Another way of addressing robustness with formal verification would be to prove formally that the precondition is satisfied in all calling contexts. It is yet to be determined if all callers should be then formally verified, or if a subset of these proofs would be sufficient.

**Acknowledgement** We are grateful to the anonymous referees for their useful comments, to Robert Dewar and Jérôme Guitton for discussions which helped shape this paper, and to Ben Brosgol for his detailed review of the manuscript.

# References

[1] `http://www.open-do.org/projects/hi-lite/a-database-example/`.

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

[3] Alfa Reference Manual. Technical report, 2011. `http://www.open-do.org/wp-content/uploads/2011/12/alfa_rm.pdf`.

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

[5] K. Beck. Simple Smalltalk testing: With patterns. Smalltalk report, Oct. 1994. `http://www.xprogramming.com/testfram.htm`.

[6] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In *SMT'08*, volume 367 of *ACM ICPS*, pages 1–5, 2008.

[7] F. Bobot, J.-C. Filliâtre, A. Paskevich, and C. Marché. Why3: Shepherd your herd of provers. In *Proceedings of Boogie 2011, the 1st International Workshop on Intermediate Language Verification*, Aug. 2011.

[8] 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. *Int. J. Softw. Tools Technol. Transf.*, 7:212–232, June 2005.

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

[10] J. J. Chilenski. An Investigation of Three Forms of the Modified Condition/Decision Coverage (MCDC) Criterion. Technical Report DOT/FAA/AR-01/18, Apr. 2001.

[11] J. Guitton, J. Kanig, and Y. Moy. Why Hi-Lite Ada? In *Proceedings of Boogie 2011, the 1st International Workshop on Intermediate Language Verification*, Aug. 2011.

[12] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002.

[13] 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.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

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

[16] RTCA. Software considerations in airborne systems and equipment certification. Document RTCA DO-178B, 1992.

[17] E. 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.

[18] 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.