Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

SPARK 2014 and GNATprove

A Competition Report from Builders of an Industrial-Strength Verifying Compiler

Duc Hoang[1], Yannick Moy[2], Angela Wallenburg[3], Roderick Chapman[3]

[1] Ecole Polytechnique Fédérale de Lausanne, e-mail: duc.hoang@epfl.ch
[2] AdaCore, e-mail: yannick.moy@adacore.com
[3] Altran UK, e-mail: {angela.wallenburg, rod.chapman}@altran.com

Received: date / Revised version: date

Abstract. Extensive and expensive testing is the method most widely used for gaining confidence in safety-critical software. With a few exceptions, such as SPARK, formal verification is rarely used in industry due to its high cost and level of skill required. The grand challenge of building a *verifying compiler* for static formal verification of programs aims at bringing formal verification to non-expert users of powerful programming languages. This challenge has nurtured competition and collaboration among verification tool builders; an example is the VerifyThis competition [HKM13]. In this paper we describe our approach to popularising formal verification in the design of the SPARK 2014 language and the associated formal verification tool GNATprove. In particular, we present our solution to combining tests and proofs, which provides a cost-competitive way to develop software to standards such as DO-178. At the heart of our technique are executable contracts, and the ability to both test and prove those. We use running examples from the VerifyThis 2012 competition and discuss the results of using our tools on those problems.

1 Introduction

High quality software, or low defect software, is often costly to develop. This is a common experience in safety-critical systems development, whether the development is driven by standards such as DO-178, or by any other need to create highly reliable and long-lived software. Today, extensive and expensive testing is the primary method used to gain confidence in such software.

The computing research community has been occupied for decades with the grand challenge of (building) *the verifying compiler* [Hoa03]:

> A verifying compiler uses mathematical and logical reasoning to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the code of the program. The compiler will work in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components.

As predicted in [Hoa03], this grand challenge has called for co-operation among research teams and it has encouraged and benefited from competition. An example of such beneficial competition is the VerifyThis competition, which is the context of this report.

Great strides have been made in proof automation. At present, so-called "push-button" static program verification is achievable for some substantial classes of industrial programs. New programming language features for program verification have been explored and theoretical models of complex existing language features have been devised to increase the reasoning capabilities for mainstream programming languages. See [HLL+12] for many insights in the programming language approach to program verification. However, there are still many remaining challenges before we can expect verifying compilers to be as widely used as testing.

Formal software verification has been successfully applied and scaled to industrial projects [WLBF09, CDH+09]. While many case studies have been successful, few formal methods have reached the take-up and maturity level where industrial non-experts continue to use the method for project after project, and where this formal method is a permanent part of the business of industrial software development. There are some notable exceptions: for example, the SPARK language and toolset for static verification has been applied for many years in on-board

aircraft systems, control systems, cryptographic systems, and rail systems [Bar12b, O'N12].

We identify two main hurdles that currently hinder the take-up of verifying compiler technology:

1. difficulty in reaching non-expert users, and
2. lack of convincing cost-benefit argument.

In this paper we will describe our approach to solving these two problems in the design of the SPARK 2014 language, a subset of Ada 2012 designed for formal verification, and the associated verification tool GNATprove. We will use running examples from the VerifyThis 2012 competition and discuss the results of using our tools on those problems.

This paper is organised as follows: first we describe the key language features of SPARK 2014, which is a complete update of the SPARK 2005 language and tools. SPARK 2014 has been designed with many lessons learned from the programming language and verification community, and naturally from experiences in industrial use of SPARK 2005. Then in Section 3 we describe our unique integration of testing and proving, which was developed in the collaborative research project Hi-Lite [CKM12] between Altran (formerly Praxis), AdaCore, and Inria. In the resulting new tool architecture the compiler and the verifier are based on the same front-end [KSD12]. In Section 4 we describe specific language features that enables specifications to be written more naturally. In Section 5, we present GNATprove, our formal verification tool, and in Section 6 its results in the VerifyThis 2012 competition. Finally we discuss ongoing work in Section 7 and conclusions in Section 8.

2 Key Language Features for Verification

All the functional specification features in SPARK 2014 are executable. This means that their verification can be performed either dynamically, by running the program, or statically with a dedicated formal verification tool.

2.1 Ada 2012

Ada 2012 introduced new language features for facilitating the specification of programs [Bar12a], many of which were inspired from the corresponding features in SPARK 2005. In the following, we describe some of these which we used in the solutions for the VerifyThis 2012 competition challenges.

The most useful of these new features is without doubt the preconditions and postconditions popularised by the Design-by-Contract approach [Mey88]. In challenge 1 (the longest common prefix problem) we can for example specify that function `LCP` expects arguments within bounds, and that it returns a bounded result. Note the use of the new *aspect* syntax in Ada 2012, in which the declaration of `LCP` is followed by keyword `with` and a list of aspects, each of which specifies a property of the declared entity, such as pre- and post-condition.

```
function LCP (A : Text; X, Y : Integer) return Natural with
  Pre  => X in A'Range and then Y in A'Range,
  Post => LCP'Result in 0 .. Index'Last;
```

In the postcondition of a function, the new attribute `Result` is used to refer to the result of the function. The attribute `Last` represents the last value (upper bound) in the range of type `Index`. Another new attribute `Old` is used to refer to the value of some variables on entry to a subprogram. These were used in solving challenge 2 (the prefix sum) of the VerifyThis competition. Note that we use the short-circuit Boolean operator `and then` for which the second argument is only executed or evaluated if the first argument is not enough to determine the value of the expression. This is helpful for example in producing valid specifications involving partial functions, like `Y /= 0 and then X / Y`.

Writing specifications is made easier by new expression forms in Ada 2012. If-expressions and case-expressions are the expression forms which correspond to the usual if-statements and case-statements. An if-expression without else-part `(if A then B)` expresses a logical implication of `B` by `A`. Quantified expressions `(for all X in A)` and `(for some X in A)` correspond to the mathematical universal and existential quantifications (see Section 4 for more information about the domain of bound variables). Expression functions define a function with a single expression, like in functional programming languages. As expression functions can be part of the specification of programs (contrary to regular function bodies), they provide a powerful way to abstract complex parts of specifications.

2.2 SPARK 2014

The new version of SPARK is based on the features of Ada 2012. There are also new features added, some of which are inspired from SPARK 2005. An example is the loop invariant.

Like preconditions and postconditions are essential new features of Ada 2012 for specification, the loop invariant pragma is essential in SPARK 2014. A loop invariant can be inserted anywhere in the main list of statements in a loop, and it expresses the cumulated effect of the loop up to that point. For example, here is the loop invariant used in challenge 1:

```
pragma Loop_Invariant
  (for all K in 0 .. L - 1 => A (X + K) = A (Y + K));
```

Note that a loop invariant in SPARK has a slightly different semantics to the classic loop invariant introduced by Hoare [Hoa69]. A classic loop invariant has to hold when reaching the loop, at the start of each iteration of the loop, and when exiting the loop. A SPARK loop invariant only has to hold when execution reaches the

corresponding program point. Essentially the loop invariant is like an assert except it also acts as a cut point in formal verification. A cut point means that the prover is free to forget all information about modified variables that has been established within the loop. Only the given Boolean expression is carried forward. For formal verification in SPARK, checks are generated for initialisation and preservation of the loop invariant, similarly to the classic approach.

In formal verification, it is very common that loop invariants compare the value of a variable at loop entry and at the n^{th} iteration of the loop. To enable such specifications, SPARK 2014 introduces the `Loop_Entry` attribute, which can be applied to such a variable. We have used that feature in our solution to the prefix sum challenge, see Section 6.

A loop variant pragma has been defined in SPARK 2014, to express a quantity varying monotonically at each iteration of the loop. As loop invariants, a loop variant can appear anywhere in the main list of statements in a loop. For example, here is the loop variant used in the longest common prefix challenge:

```
pragma Loop_Variant (Increases => L);
```

Note that this variant does not take the usual non-negative decreasing argument. Instead, it takes a list of increasing or decreasing integer values, bounded by their type in Ada, and the overall order over this list is the lexicographic order combined with individual directions. In the example above, there is only one element in the list, so it should increase at each run through the loop. Like for loop invariants, the point where this increase matters is the program point where the loop variant appears in the code. As in the classic case, the value of the variant is compared against the value at the corresponding program point in the previous iteration of the loop.

Subprogram contracts can become quite large, even with the use of (expression) functions to abstract common parts of contracts. Therefore, SPARK 2014 allows the definition of contracts by cases, similar to behaviours in JML [BCC+05]. For example, the contract of `LCP` can state separately sub-contracts for the cases where the elements at `X` and `Y` are different, or `X` and `Y` are equal. This contract may be used instead of or in addition to a precondition and a postcondition.

```
function LCP (A : Text; X, Y : Integer) return Natural with
  Contract_Cases =>
    (A (X) /= A (Y) => LCP'Result = 0,
     X = Y          => LCP'Result = A'Last - X + 1,
     others         => LCP'Result > 0);
```

Note that the cases above are disjoint and complete, as expected given the SPARK design goals: one and only one case should be applicable at every call. The presence of the `others` case ensures the completeness here. The SPARK tools ensures disjointness and completeness of the contract cases by augmenting the precondition with a check that exactly one of the conditions of the contract case list is satisfied, and conjoining the postcondition with conditional expressions representing each of the contract cases.

3 Integrated Testing and Proving

As we have mentioned, in the development of the new generation language and toolset SPARK 2014, we have a particular focus on providing a good cost-benefit argument and on reaching non-expert users.

We will describe a few observations of what drives current practices in the industry, in order to help us with the cost-benefit argument. We will also see how progress in the research of behavioural interface specification languages [HLL+12] enables an approach where test and proof can be elegantly integrated.

3.1 Motivation: Industry Safety Standards and Testing

Industry standards and certification documents highly influence the state-of-the-practice of safety-critical software development. DO-178B [RTC92] is a document that is used as a basis for certification of airborne software by institutions such as Federal Aviation Administration (FAA) and European Aviation Safety Agency (EASA). Though DO-178B is for avionics, it is often used in other safety-critical sectors as well. It is regarded as very successful within the avionics industry itself; since its introduction in 1992 no commercial aircraft fatality has been attributed to DO-178B-certified software. The majority of objectives in DO-178B consider verification. DO-178B is non-prescriptive with regards to programming languages, software tools, particular development processes etc. The development of requirements-based tests is mandatory, including normal range tests cases and robustness (abnormal range) test cases. The standard requires verification of both high-level requirements and low-level requirements. Three levels of testing are defined in DO-178B: hardware/software integration testing, software integration testing, and low-level testing – all based on requirements. The test cases must fully cover the code and all exercised code should be traceable to requirements. This requires coverage analysis against specified criteria, such as MC/DC. Such structural low-level testing, together with robustness testing is expensive. For example, a large cost is associated just for collecting and verifying output of these tests. Our goal is to reduce this cost, while still meeting the objectives prescribed by the standard.

Formal methods can help to verify that no anomalous behaviour will occur, for example they can be used to prove the absence of run-time errors. Formal methods can also be used to show compliance between a program's actual and specified behaviour. Furthermore, a proof of program correctness is comparable to exhaustive analysis achieving 100% coverage. Though formal

methods and other kinds of verification are recognised as beneficial, they do not contribute to DO-178B certification credit, and their use has had to be justified by other means. However, DO-178C [RTC11], the recently released successor and replacement of DO-178B, allows some of the prescribed testing activities to be replaced by formal methods. From DO-178C [RTC11]:

> The use of formal methods is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to establishing the correctness and robustness of a design.

Formal methods are complementary to testing, and may find faults that are not detected by testing, but they cannot establish verification evidence for the target hardware. Therefore testing on the target is still required. However, formal analysis of source code can be used to show compliance with the low-level requirements. DO-178C requires an argument for property preservation between the source code and the object code for those properties that have been verified formally at the source level. Since formal program verification and testing are complementary, we would like to use each method where it is most efficient. For this we need to make sure that the combination is at least as strong as testing alone.

3.2 Executable Contracts

Programming using contracts is a way to organise your code. By stating a precondition and a postcondition of a subprogram as for example in Section 2, we assign responsibilities. The subprogram is responsible for the postcondition to be met, as long as it is called under the assumptions of its precondition, which it relies upon. Similarly, a caller of this subprogram is responsible for ensuring that the precondition of the subprogram holds, before calling it. It can then rely on the postcondition of the called program upon return. This programming discipline encourages a modular design of the software. Furthermore when verification is concerned, this can be done in a modular fashion as well. Modularity gives the flexibility to perform verification during development, rather than waiting until after integration.

In addition to user-defined contracts we also have implicit contracts given for example by a strong type system such as Ada's and the signatures of subprograms. For example there is an implicit precondition that subprogram parameters have values within their types and likewise an implicit postcondition that guarantees that output parameters and function results have values allowed by their type. As an example, consider what difference it makes to the user-supplied contract, if you would use the type `Natural` instead of `Integer` for the `Input` parameter of procedure `Sqrt`:

```
procedure Sqrt (Input : Integer; Res: out Natural) with
  Pre  => Input >= 0,
  Post => (Res * Res) <= Input and then
          (Res + 1) * (Res + 1) > Input;
```

In this case the implicit precondition would be sufficient and a user-supplied precondition would not be necessary. Since there is a cost associated with users writing contracts, there is a benefit in designing the language so that the basic language features already provide a rich default contract.

The notion of preconditions and postconditions was first introduced by Hoare [Hoa69], and later reinvented as Design-by-Contract by Meyer [Mey88]. Traditionally, contracts have been interpreted quite differently depending on whether used for formal program verification or for run-time assertion checking. For formal program verification, assertions have typically been interpreted as formulae in classical first-order logic. This is not consistent with the run-time assertion checking semantics.

Much effort has been spent the last decades to popularise formal program verification. [Cha10] did something as unusual as surveying practitioners, to find that they prefer run-time assertion checking semantics. Furthermore he developed a semantics to allow formal executable contracts *i.e.,* compatible with run-time assertion checking, in the hope that those users who are already annotating their code with assert statements, could more easily be convinced to start writing contracts. Ada 2012 and thus SPARK 2014 have such semantics.

Ada 2012 assertions have an executable semantics prescribed by the standard, in which an assertion may fail due to a run-time error during its evaluation, and types are the machine ones (bounded integers, floating-point or fixed-point). This is in contrast to SPARK 2005 assertions, that have a logical semantics in which run-time errors are ignored, and types are the mathematical ones (infinite precision integers and reals). We have reconciled these views in SPARK 2014 so that assertions have both an executable and a logical interpretation. Furthermore, these interpretations are consistent in the sense that if a run-time error could occur during execution, there will be a corresponding verification condition during formal verification with the SPARK tools.

As users can execute contracts, they can also debug them like code, and test them when formal verification is too difficult to achieve. Furthermore, there is an advantage in keeping the annotation language the same, or almost the same, as the programming language: users don't have to learn one more language. If the contract is also formal, the entry barrier to formal program verification can be lowered by making it available to those who would write executable contracts.

3.3 Mixing Test and Proof

Low-level requirements, in DO-178 terms, are typically expressed in natural language at subprogram or unit level. Formal executable contracts can be used to express

those requirements at the subprogram level. Thanks to the same semantics for test and proof as we have just discussed, we can use either (or both) of test and proof to verify a subprogram. Thanks to modular verification, it is then possible to mix test and proof to use the verification method that is most cost effective for each module. If the chosen method is testing the benefits of formal executable contracts are:

- Low-level requirements expressed as contracts
- Successful execution of postcondition → test successful
- No need to collect and verify output

If the chosen verification method for a subprogram is proof, the benefits of formal executable contracts are:

- Low-level requirements expressed as contracts
- Successful proof of postcondition → low-level requirement verified for all input
- Approach allowed by DO-178C formal methods supplement
- Proving process faster when annotations can be executed (debugging failed proof attempts is very time-consuming)

But what about soundness of the mixed approach? When some low-level requirements are tested and some are proved, in DO-178C-terms the combination needs to be as good as if all low-level requirements were tested.

Central to modular verification is the statement of assumptions and guarantees. For global correctness, *all* subprograms must establish their postconditions (under the assumption of their preconditions), and for *all* calls to a subprogram, its precondition must be verified by the caller before the call.

In our mixed approach of test and proof, we have these same global correctness requirements as usual for modular verification. The difference is that some subprograms may be tested and some be proved, and we must still make sure that all assumptions are verified. Let us consider the two cases: 1) a tested subprogram `T` calls a proved subprogram `P`, and 2) a proved subprogram `P` calls a tested subprogram `T`. In the first case, when verifying `T` we must make sure that the precondition of `P` is established. When testing `T`, this can be done by executing the precondition of `P`. In the second case, the correctness of `P` relies on that after having verified the precondition of and then called `T`, then `T` should return in a state where the postcondition holds. This assumption on `T` should be verified when testing `T` by executing the assertion that its postcondition holds. Both of these verification cases are only possible because we have executable contracts.

Although we don't describe it here, the same verification by testing of assumptions made during formal verification needs to be done for implicit contracts, related to initialization of in-type values and non-aliasing. We have implemented these additional verifications in the GNAT compiler through dedicated switches that the user can set [CKM12].

3.4 Compiler Implementation

For SPARK 2014 we are building the verification tools based on the front-end of the GNAT compiler. [KSD12] describes and discusses this architecture. One advantage here is that the compiler inserts these additional checks needed to verify at run time assumptions made during proof. Another advantage is that formal verification has access to the precise configuration of the target platform used for compilation (size of integers, endianness, *etc.*), which facilitates performing proofs that depend on the target.

3.5 Related Work on Combined Test and Proof

The combination of dynamic and static analysis has received a fair amount of attention during the last decade. Some examples include combinations of symbolic and concrete execution: DART [GKS05], CUTE [SMA05] and EXE [CGP+06], contract and invariant inference tools that use testing such as Daikon [EPG+07] and QuickSpec [CSH10], and finally HipSpec [CJRS13], which uses a combination of automatic inductive theorem proving and testing to generate lemmas. Other related systems included [BFL+11] whose compiler also emits run-time checks for contracts, as well as the KeY Unit Test Generator [BGTY11]. It would be interesting to explore some of these approaches in SPARK in the future.

The work perhaps most similar to our combination of test and proof is the verification environment Eve [TFNM11] for Eiffel. It also uses a modular contract-based approach where postconditions serve as test oracles and some modules can be verified by test and some by proof. One notable difference is that in Eve the semantics for integers is different during test and proof. We will discuss this topic further in the next section. Our solution is unique in that it has a practical focus on replacing testing by a mix of testing and proving that is sound in DO-178C terms.

4 Choosing the Right Semantics for Integers

As already mentioned, there has traditionally been a difference in interpretation of contracts depending on whether used for formal verification, or for run-time assertion checking. One area where this difference is of practical significance is in the semantics for integers. An effect of having the same semantics in assertions as in the program code is that run-time exceptions must be considered, and avoided, in the assertions as well as in the program. As an example, consider a programmer who wants to state a precondition that the addition of two numbers fits in a desired integer type:

```
Pre => X + Y in Some_Integer_Type
```

The problem is that `X + Y` itself can cause an arithmetic overflow, and worse this is not relevant to the correctness of either the program or the specification. When executable semantics are used for integer operations in assertions, there will therefore in practical formal verification tools be a large number of extra proof obligations that do not point to real issues either in the program code or the assertions. In traditional assertion languages aimed for formal verification, mathematical universal integers are used for such operations and proof can safely proceed without involving the user. Furthermore, user studies [Cha04] show that users who are otherwise happy with run-time assertion checking semantics, still prefer mathematical semantics for intermediate integer operations in assertions.

From a user perspective, mathematical semantics for integer operations in assertions is a good solution. However, for our hybrid verification argument described in the previous section to work, it is mandatory to have the same semantics for test and proof. This means that the compiler needs to implement the ability to execute mathematical integer operations. Fortunately, it has been possible to implement a solution for this in the GNAT compiler, which we are building our verification tools on. Also, the Ada standard permits this approach by not requiring that the program issues an overflow error when an intermediate value does not fit into the base type, providing the correct result is computed.

Existing users of SPARK 2005 and potential new users of SPARK 2014 together cover a wide spectrum of strong preferences regarding this issue. Existing users of SPARK 2005 who perform formal verification of absence of run-time errors for their programs want the traditional mathematical integers in assertions and the standard Ada semantics for the program. Another user wants the same executable semantics in assertions and program code, and wants to leave the assertions in the shipped code. Many users could not have executed code that uses a library for unbounded integers because of certification and performance reasons, *etc.* Because of these strong needs, we provide three alternative overflow checking modes in GNAT and GNATprove:

1. Strict mode: normal overflow checks
2. Minimized mode: larger base type (64bits) used when needed
3. Eliminated mode: use bignum library in the remaining cases

In the second and third cases the compiler performs a simple static analysis and decides if a larger machine integer, or even an unbounded one, is needed to fit the intermediate value of an arithmetic operation. In minimized mode, if the compiler static analysis decides that the value will fit in the 64 bit type, it will use a variable of the 64 bit type for the intermediate result. If it finds that the value may overflow the 64 bit type, a check will be emitted. These three modes for the user to choose from, gives a flexible solution. The choice is independent for assertions and code. However, the same choice for execution and formal verification is required for our hybrid verification with test and proof to be sound.

There is a potential source of confusion in providing several overflow checking modes but this is managed with SPARK language profiles. SPARK users can chose pre-defined profiles and customise their own profile to prohibit particular language features, and enforce tool settings such as overflow semantics, according to project-specific constraints and regulations.

5 Making Automatic Verification Work

GNATprove uses the Why3 platform [BFPM11] to generate verification conditions (*VC*s) and call provers. It can target as many output formats and automatic or manual provers as the Why3 platform allows (many!), but we focus on the automatic proof of VCs through the use of the SMT prover Alt-Ergo [BCCL08], which is distributed with GNATprove. There are two steps to make automatic verification work: first make it possible, then make it efficient.

The choice of source programming language is essential to make automatic verification possible. On the one hand, it should proscribe those features which make automatic verification impossible or hinder scalability, completeness and efficiency on industrial code-bases. On the other hand, it should contain enough features which facilitate the expression of specifications. The former is obtained by restricting SPARK 2014 to a subset of Ada 2012 without pointers, exceptions, aliasing, and side-effects in expressions. The latter is obtained by the features introduced in Ada 2012 and SPARK 2014. Ada 2012 was specifically designed to include preconditions, post-conditions, type invariants, *etc.* so that users can specify arbitrarily complex invariant properties on the data and control of their programs, and test these properties at run time. As mentioned, SPARK 2014 further adds loop invariants, loop variants, *etc.* so that a user can formally prove these properties.

Efficient formal verification relies on a subtle coordination between the VC generator and the prover, so that the VCs produced can be efficiently proved. GNATprove relies heavily on the features of the Why3 language [GKM11] to produce provable VCs. For example, the VCs are kept small by translating the semantic dependencies between entities at the Ada source code level into syntactic inclusions between modules at the Why3 intermediate code level, and by using the abstraction feature in Why3 for the intermediate code that checks for absence of run-time errors. This ensures that the generated VCs only contain relevant definitions and axioms. As another example, one can choose to produce fewer but more complex VCs: the default in GNATprove is

that a VC accounts for all paths leading to an assertion, using an efficient computation [Lei05], instead of generating one VC for each path leading to an assertion (also available in GNATprove as an option). The choice of axiomatisation of Ada data types (integer types, enumeration types, record types, array types, *etc.*) in Why3 also has a significant effect on the provability of VCs. We have tuned these axiomatisations to better suit the mechanisms inside SMT provers like Alt-Ergo. Similarly, we have tailored the axiomatisation for a generic library of containers [DFM11] to SMT provers.

Finally, modular verification based on pre- and post-conditions can very easily exploit multi-core architectures, as the generation of VCs for different units, or the proof of different VCs, can both be run in parallel. Typically, projects contain hundreds of units, and lead to the generation of thousands of VCs, which can be run by GNATprove on as many cores as are available. Note also that GNATprove uses file timestamps to avoid regenerating VCs for units which have not been updated, and file hashes to avoid re-proving VCs that have already been proved. This is crucial when developing either the code or the associated annotations, to avoid unnecessary rework.

6 VerifyThis Competition

In this section, we will describe our solutions to two of the challenges from the 2012 VerifyThis competition. The organizers report of the competition [HKM13], includes the full descriptions of the challenges. In this section we will restate the descriptions of the verification tasks briefly. The first challenge is the longest common prefix problem, `LCP`. The second challenge, `PrefixSum`, is a binary tree summation algorithm implemented inplace on an array. For the latter we have one complete fixed-size solution suitable for automatic verification using the current version of SPARK 2014 and GNATprove. Furthermore, we have specified a general version of the `Upsweep` procedure of the second challenge initially using SPARK 2005 as a reference.

6.1 How the Solutions Were Produced

A solution to challenge 1 and an initial solution to challenge 2 was submitted during the VerifyThis conference. In the aftermath, two parallel activities were carried out. One activity was to tune and develop settings to make automatic verification work better in GNATprove. This is described in sections 5 and 6.5. The other activity was to generalise the solution to challenge 2 to provide a more elegant specification, described in Sect. 6.6. The general specification was produced both in SPARK 2005 and SPARK 2014, partly serving as a useful mini case study to compare the new and old versions of the tools during development of the new tools. Unfortunately we did not find the time to explore a full general version of challenge 2 or challenge 3.

6.2 Challenge 1: Longest Common Prefix

Longest Common Prefix (LCP) is a text query problem that can be informally specified as follows:

- Input: an integer array `a`, and two indices `x` and `y` into this array
- Output: length of the longest common prefix of the subarrays of `a` starting at `x` and `y` respectively.

Prove that your implementation complies with a formalized version of the above specification.

6.3 Solution to Challenge 1: Longest Common Prefix

The longest common prefix solution can be readily coded in SPARK 2014 as follows:

```
subtype Index is Positive range 1 .. 1_000_000;
type Text is array (Index range <>) of Integer;

function LCP (A : Text; X, Y : Integer) return Natural with
  Pre  => X in A'Range and then Y in A'Range,
  Post =>
    (for all K in 0 .. LCP'Result - 1 =>
      A (X + K) = A (Y + K))
      and then
    (X + LCP'Result = A'Last + 1
      or else Y + LCP'Result = A'Last + 1
      or else A (X + LCP'Result) /= A (Y + LCP'Result));

function LCP (A : Text; X, Y : Integer) return Natural is
   L : Natural;
begin
   L := 0;
   while X + L <= A'Last
     and then Y + L <= A'Last
     and then A (X + L) = A (Y + L)
   loop
      pragma Loop_Invariant
        (for all K in 0 .. L - 1 =>
          A (X + K) = A (Y + K));
      pragma Loop_Variant (Increases => L);
      L := L + 1;
   end loop;
   return L;
end LCP;
```

The input specification that parameters `X` and `Y` are indices in the array parameter `A` can be expressed as a precondition involving the Ada attribute `'Range`, and membership tests `X in ...` The output specification that the result is the length of the longest common prefix starting at `X` and `Y` can be expressed as a postcondition in two parts, using the Ada 2012 attribute `'Result` to express the function result:

- A quantification stating that the subarrays of `A` of length `LCP'Result` starting at `X` and `Y` are equal.

– A disjunction of cases stating that either one of the two subarrays reaches the end of array `A`, or the elements following the two subarrays in `A` are different. Note here that the use of the lazy Boolean connective `or else` is compulsory to make sure that `X + LCP'Result` and `Y + LCP'Result` are within the bounds of `A` when accessing `A` in the last line of the postcondition.

Running GNATprove on this code without loop invariant or loop variant results in the generation of 13 VCs: 1 VC for the postcondition, and 12 VCs for all run-time checks (6 array index checks, 5 numeric overflow checks, 1 subtype range check). All VCs related to run-time checks are proved. These VCs represent both checks in the code and checks in assertions for the array accesses in the postcondition. The VC for the postcondition is not proved, due to the presence of a loop in the body of `LCP`.

Proving the postcondition requires the insertion of a loop invariant in the body of `LCP`, which expresses that the subarrays of `A` of length `L` starting at `X` and `Y` are equal. Since `L` is the value returned by `LCP`, this loop invariant matches the first part of the postcondition when the loop terminates. As expected, the postcondition is proved with this additional loop invariant. GNATprove generates 4 additional VCs to prove that the loop invariant initially holds at the first iteration through the loop, that it is maintained by subsequent iterations, and that the two array accesses in the loop invariant expression are within bounds. All 4 additional VCs are also proved.

Finally, proving termination of `LCP` requires the insertion of a loop variant in the body of `LCP`, which expresses that the value of `L` always increases between two consecutive iterations through the loop. Since `L` is of a bounded type (the scalar type `Natural` of natural numbers), it cannot be infinitely incremented without failing a run-time check. Since we have already proved that no run-time check fails in `LCP`, proving the variant proves the termination of `LCP`. The corresponding VC is proved by GNATprove.

The final version of `LCP` is proved in 6s on a laptop with 4G RAM and a 3GHz processor.

6.4 Challenge 2: Prefix Sum

This is an abbreviated description of the challenge from the organizers report of the competition [HKM13], excluding the provided reference implementations and figures.

Background: The concept of a prefix sum is very simple. Given an integer array a, store in each cell $a[i]$ the value $a[0] + \ldots + a[i-1]$. For example, the prefix sum of the array [3, 1, 7, 0, 4, 1, 6, 3] is [0, 3, 4, 11, 11, 15, 16, 22]. Prefix sums have important applications in parallel vector programming, where the workload of calculating the sum is distributed over several processes. We will verify a sequentialized version of a prefix sum calculation algorithm.

Algorithm: We assume that the length of the array is a power of two. This allows us to identify the array initially with the leaves of a complete binary tree. The computation proceeds along this tree in two phases: upsweep and downsweep. During the upsweep, which itself proceeds in phases, the sum of the children nodes is propagated to the parent nodes along the tree. A part of the array is overwritten with values stored in the inner nodes of the tree in this process. After the upsweep, the rightmost array cell is identified with the root of the tree. As preparation for the downsweep, a zero is inserted in the rightmost cell. Then, in each step, each node at the current level passes to its left child its own value, and it passes to its right child, the sum of the left child from the upsweep phase and its own value.

Verification Task: We provide an iterative and a recursive implementation of the algorithm. You may choose one of these to your liking.

1. Specify and verify the upsweep method. You can begin with a slightly simpler requirement that the last array cell contains the sum of the whole array in the post-state.
2. Verify both upsweep and downsweep – prove that the array cells contain appropriate prefix sums in the post-state.

If a general specification is not possible with your tool, assume the length of array is 8.

6.5 Solution to Challenge 2: Prefix Sum

We have chosen to implement an iterative version of prefix sum instead of a recursive one, which better matches the constraints commonly found in critical embedded software where recursion is usually not allowed. In order to make automatic proof easier, we fix the length of the array to 8. The complete solution is quite long (186 lines of code, not counting empty lines), so we only show here selected parts. The initial solution without annotations is straightforward and only 50 lines long. As an example, here is the implementation of procedure `Upsweep`:

```
procedure Upsweep (A : in out Input;
                   Output_Space : out Positive) is
   Space : Positive := 1;
   Left  : Natural;
   Right : Natural;
begin
   while Space < A'Length loop
      Left := Space - 1;
      while Left < A'Length loop
         Right     := Left + Space;
         A (Right) := A (Left) + A (Right);
         Left      := Left + Space * 2;
      end loop;
```

```
      Space := Space * 2;
   end loop;
   Output_Space := Space;
end Upsweep;
```

The postcondition of procedure `Upsweep` states that the array parameter `A` is put in an intermediate form w.r.t. its initial value (denoted `A'Old`). The contract of procedure `Downsweep` states that it takes as input an array parameter `A` in an intermediate form w.r.t. a `Ghost` array parameter, and that it outputs via the same parameter `A` the desired prefix sums of array `Ghost`. By calling in sequence `Upsweep` and `Downsweep` on an array `A`, with the initial value of `A` passed as `Ghost` parameter, a caller performs the desired in-place modification of `A`. Although SPARK 2014 does not yet support ghost parameters, which are only used for proofs, this is the role of parameter `Ghost` here, hence its name.

```
procedure Upsweep (A : in out Input;
                   Output_Space : out Positive) with
  Pre  => All_Elements_In (A, Maximum),
  Post => All_Elements_In (A, 8 * Maximum)
    and then Output_Space = 8
    and then Intermediate_Form (A, A'Old);

procedure Downsweep
  (Ghost : Input; A : in out Input;
   Input_Space : in Positive)
with
  Pre => All_Elements_In (Ghost, Maximum)
    and then All_Elements_In (A, 8 * Maximum)
    and then Input_Space = 8
    and then Intermediate_Form (A, Ghost),
  Post =>
    A (0) = 0
      and then
    A (1) = Ghost (0)
      and then
    A (2) = Ghost (0) + Ghost (1)
      and then
    A (3) = Ghost (0) + Ghost (1) + Ghost (2)
      and then
    A (4) = Ghost (0) + Ghost (1) + Ghost (2) + Ghost (3)
      and then
    A (5) = Ghost (0) + Ghost (1) + Ghost (2) + Ghost (3)
          + Ghost (4)
      and then
    A (6) = Ghost (0) + Ghost (1) + Ghost (2) + Ghost (3)
          + Ghost (4) + Ghost (5)
      and then
    A (7) = Ghost (0) + Ghost (1) + Ghost (2) + Ghost (3)
          + Ghost (4) + Ghost (5) + Ghost (6);
```

The function `Intermediate_Form` gives the exact relationship between the initial value of the array (parameter `B` below) and its intermediate value between the calls to `Upsweep` and `Downsweep` (parameter `A` below). We define it as an expression function in Ada 2012, which has the benefit that GNATprove automatically generates a postcondition for `Intermediate_Form` equivalent to its body. We also give a precondition to `Intermediate_Form` to prove that its evaluation cannot fail run-time checks (overflow checks and index checks here).

```
function Intermediate_Form (A, B : Input) return Boolean
  with Pre => All_Elements_In (A, Maximum * 8)
          and then All_Elements_In (B, Maximum);

function Intermediate_Form (A, B : Input) return Boolean is
  (for all K in A'Range =>
     (if (K + 1) mod 8 = 0 then
        A (K) = B (0) + B (1) + B (2) + B (3) +
                B (4) + B (5) + B (6) + B (7)
     elsif (K + 1) mod 4 = 0 then
        A (K) = B (K) + B (K-1) + B (K-2) + B (K-3)
     elsif (K + 1) mod 2 = 0 then
        A (K) = B (K) + B (K-1)
     else
        A (K) = B (K)));
```

Note that the contracts of all previous procedures and functions contain calls to function `All_Elements_In`, which returns `True` if all elements of an array are bounded in absolute value, which we also define as an expression function:

```
function All_Elements_In (A : Input; Max : Positive)
return Boolean is
   (for all K in A'Range => A (K) in -Max .. Max);
```

These specifications add 62 lines to the initial 50 lines for the solution. To prove them with GNATprove, we add 84 more lines for loop invariants and loop variants. As an example, here are the loop invariant and loop variant for the inner loop of procedure `Upsweep`. Inside the loop invariant, `A'Loop_Entry` denotes the value of `A` on entry to the loop.

```
pragma Loop_Invariant (
  (Left + 1) mod Space = 0
    and then
  All_Left_Elements_In (A, Left, Space * 2 * Maximum)
    and then
  All_Right_Elements_In (A, Left - 1, Space * Maximum)
    and then
  (Left + 1) mod (Space * 2) = Space
    and then
  (if Left >= A'Length then Left = 8 or Left = 9)
    and then
  (for all K in A'Range =>
    (if K in A'First .. Left - Space
       and then (K + 1) mod (2 * Space) = 0
     then
       A (K) = A'Loop_Entry (K) + A'Loop_Entry (K - Space)
     else
       A (K) = A'Loop_Entry (K))));
pragma Loop_Variant (Increases => Left);
```

The final version of prefix sum is partially proved (78 VCs proved, 8 VCs unproved) in 138s on a multi-core laptop with 24G RAM at 2.4GHz (to be able to prove VCs in parallel). These 8 unproved VCs are either loop invariants or postconditions, which are not proved automatically due to the use of the `mod` operator, currently not well handled in the underlying automatic prover. We have checked all of them manually, and written an expanded version of loop invariants and intermediate function where all occurrences of the `mod` operator are replaced by an explicit enumeration of the possible cases for all indexes, which GNATprove is able to fully prove automatically.

6.6 General Prefix Sum

As mentioned, we chose to implement an iterative version of `PrefixSum` as a better fit for critical embedded software. Though the presented complete solution makes automatic proof possible with the prototype version of our tools, the specification could be improved. Here we present a general specification for the simpler property of the `Upsweep` procedure. This solution avoids explicit listing of intermediate state in the contract, because it is too detailed and hard to judge whether it is correct, and it would be impossible to use that specification scheme for a larger array.

The `Upsweep` precondition consists of two parts: 1) the starting values in the array are within a certain `Maximum`, which is specified here to be as large as it can be without causing overflows, and 2) the length of the array is a power of two. The postcondition state two observations about the algorithm: 1) the even positions are left unchanged during the upsweep, and 2) upon completion of `Upsweep`, the last element holds the sum of all the elements in the array.

We have added a constant `Tree_Depth` for the size of the tree that fits in the input array. This is to be able to neatly specify an important precondition on `Upsweep` that the size of the array is a power of 2. Otherwise the binary tree algorithm is not guaranteed to work, so we want to be explicit about this precondition. By adding this constant and precondition we ask the user to effectively fix the size, but it is done in a general fashion. It is worth noting that this precondition could have been stated without asking the user to fix the size, and perhaps more naturally as $\exists n : Integer.\ A'Length = 2^n$, but it is our experience that existential quantification is difficult for the automatic provers that we have available. In this example the problem would show for callers of `Upsweep` trying to establish its precondition, and our typical automatic prover would essentially stay busy trying to find a model for $\forall n : Integer.\ A'Length \neq 2^n$.

We have added a couple of functions to make the specification intuitive: `Is_Even`, and more importantly `Summation`, the mathematical operation for adding a sequence of numbers.

```
Tree_Depth : constant := 3;
Maximum    : constant := Integer'Last / 2 ** Tree_Depth;

function Is_Even (K : Integer) return Boolean is
  (K mod 2 = 0);

function Summation (A : Input; Start_Pos, End_Pos : Index)
return Integer with
  Pre => Start_Pos <= End_Pos;

function Summation (A : Input; Start_Pos, End_Pos : Index)
return Integer is
  (if Start_Pos = End_Pos then
     A (Start_Pos)
   else
     A (End_Pos) + Summation (A, Start_Pos, End_Pos - 1));
```

```
procedure Upsweep (A : in out Input;
                   Output_Space : out Positive) with
  Pre  =>
   (for all K in A'Range => A (K) in -Maximum .. Maximum)
      and then
   A'Length = 2 ** Tree_Depth,
  Post =>
   (for all K in A'Range =>
     (if Is_Even (K) then A (K) = A'Old (K)))
    and then
   A (A'Last) = Summation (A'Old, 0, A'Last);
```

The downside is that this improved specification does not yet verify with the current version of GNATprove, since the proof requires the addition of an axiom for the summation function. We plan to add support for such axioms within GNATprove in the future.

6.7 Useful Tool Features

The format of the competition exercised useful features of GNATprove, which helped to find errors early in the code and in the annotations. The most useful of these features is certainly the ability to execute annotations, which allows annotations to be tested and debugged as if they were code. It was used during the competition to quickly locate the reason for an unprovable VC on challenge 1: the loop exit test was using a strict comparison operator instead of the correct non-strict one. To locate the problem, the participating author simply wrote a test exercising `LCP` on an input, compiled it with run-time checks, and executed it. The raised run-time error precisely located the failing loop invariant. Although this example was simple enough to immediately understand the underlying problem, it would have been possible to use the debugger to further investigate the issue, which can be extremely useful on real industrial code. The ability to execute annotations was also useful for challenge 2, which requires the development of complex loop invariants, to quickly correct erroneous ones. This feature won the prize of user-assistance tool feature awarded by the jury of the VerifyThis competition.

Another very useful feature for these challenges was the ability to eliminate completely all possibilities of numeric overflow in annotations, as described in Section 4. This reduced the number of false alarms. It was done by setting the overflow checking mode of GNAT and GNATprove to "eliminated". While using the "strict" overflow mode results in only 10 more overflow VCs in challenge 1, which are all proved with the current annotations, it adds 60 overflow VCs in challenge 2, most of which require modifications of annotations, or addition of new annotations, to be proved. Note that this feature is compatible with the execution of annotations, as compilation also takes into account the overflow mode when compiling arithmetic expressions.

Various features of GNATprove make it very convenient to use inside an Integrated Development Environment (currently GPS, the GNAT Programming Studio,

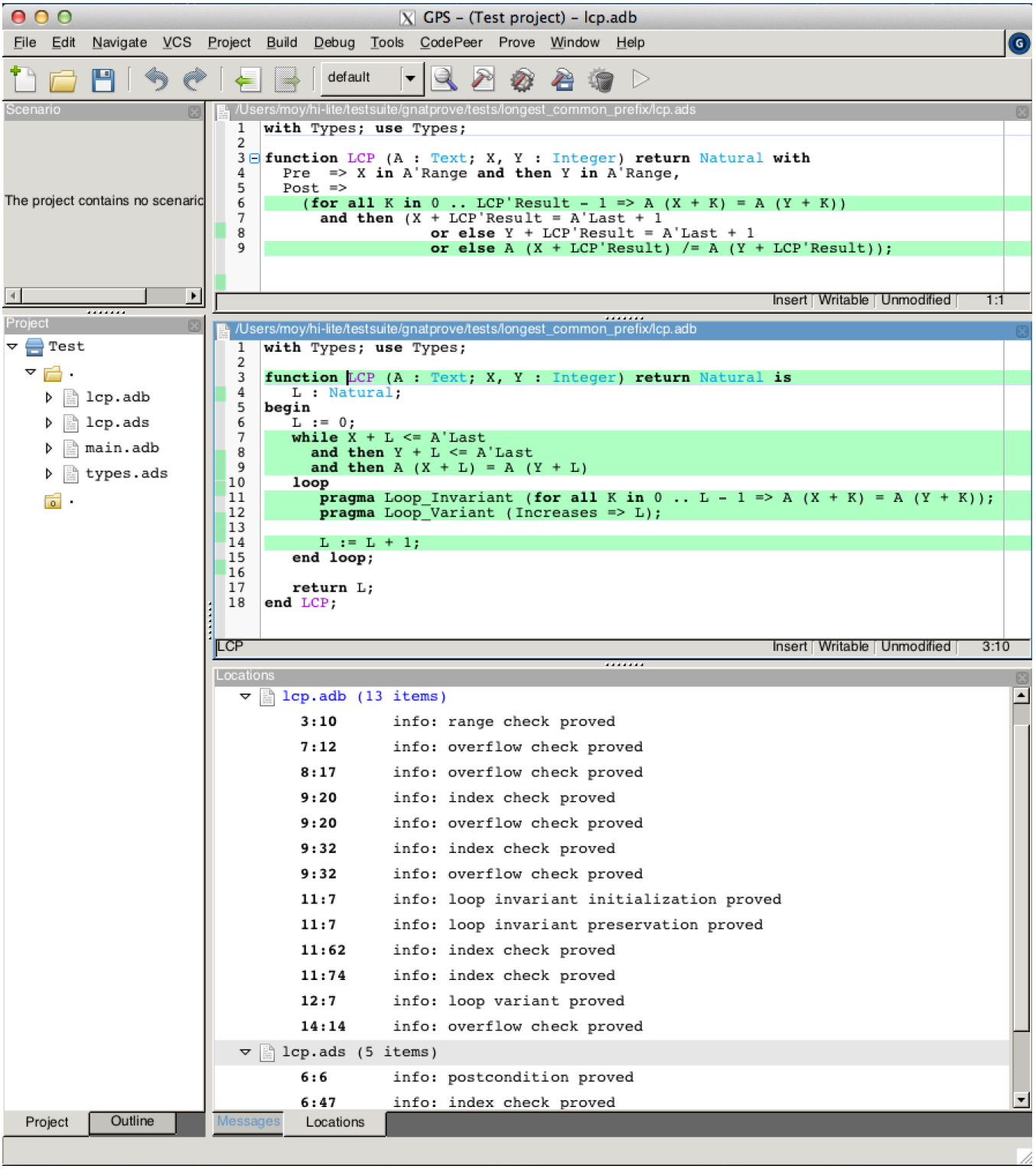

Fig. 1. Snapshot of GNATprove results on challenge 1 inside GPS

see Fig. 1). The user can choose to call GNATprove on a selected file, an individual subprogram, or even a single line of code. This was key to speeding up the modify/verify loop during the competition. When a VC is not proved, the IDE can also display the corresponding statements in the code, to help figure out why some assertions do not hold on some paths.

More generally, these challenges make use of numerous language features in Ada 2012 and SPARK 2014 that facilitate the expression of specifications: preconditions and postconditions, loop invariants and loop variants, special attributes `'Result`, `'Old` and `'Loop_Entry`, expression functions, quantified expressions and if expressions.

7 Ongoing Work

GNATprove is the result of a complete redesign of the SPARK language and associated tools, which started in 2010 with project Hi-Lite [Hi]. Altran and AdaCore are collaborating to complete this new version of SPARK by the start of 2014. On the tool side, current work focuses on flow analysis, support for investigating unproved VCs, and improvements of the SMT prover.

Flow analysis is the verification of the data dependences of subprograms. This analysis, which has always been a component of SPARK verification, is being redeveloped for SPARK 2014 based on program dependence graphs [HRB88]. An important novelty is that, while SPARK 2005 requires that the user annotates programs with data dependence contracts, they are optional in SPARK 2014, and GNATprove generates them when not present. A minimal flow analysis is always required for the soundness of proofs, while a more complete flow analysis is optional. The minimal flow analysis ensures that all variables are initialized prior to being read. For some types, their representation allow bit patterns that are outside of the values of the type. Examples of such types are Boolean represented by a Byte, floating point types, or integer types that do not span their entire base type. The implicit in-type assumption, required to deduce conclusions like $X = True \vee X = False$ where X is a Boolean program variable, is therefore not a sound assumption if X is uninitialized.

Good support in the investigation of unproved VCs is key to making formal program verification cost-effective in industry. GNATprove currently provides various solutions to that problem: the ability to execute annotations to detect errors in code and/or annotations; the display of program paths to detect errors or locate missing annotations; the possibility to call alternative provers to identify prover shortcomings. In the future, we would like to add the display of counter-examples generated by the prover, as already provided by some SMT provers [BT07, dMB08], and in Riposte [SB12], a counter-example finder for SPARK 2005.

We have been exploring two promising ways to improve the results of the Alt-Ergo SMT prover on the VCs generated in GNATprove: handling selected axiomatisations as decision procedures [DCKP13], and incrementally selecting axioms [CGS09, KvLT+12]. More work is needed to make these modes the default in GNATprove.

8 Conclusions

In this paper we have described the key elements of the design of SPARK 2014 and GNATprove. This design is based on experience over many years working with both expert and non-expert SPARK 2005 developers in industry. The design focuses particularly on improving the usability for non-expert users and providing a convincing cost-benefit argument for using formal verification in regulated industries. SPARK 2014 and GNATprove are still being implemented, and it is early days for DO-178C, but early evaluations are very encouraging, and some are documented in the case study developed by Astrium Space Transportation [LMK13].

9 Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of this report.

References

[Bar12a] John Barnes. Ada 2012 rationale. 2012. `http://www.adacore.com/knowledge/technical-papers/ada-2012-rationale/`.

[Bar12b] John Barnes. *SPARK: The Proven Approach to High Integrity Software.* Altran Praxis, 2012. third edition of the SPARK book.

[BCC+05] 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.

[BCCL08] François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing polymorphism in SMT solvers. In *SMT'08*, volume 367 of *ACM ICPS*, pages 1–5, 2008.

[BFL+11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[BFPM11] François Bobot, Jean-Christophe Filliâtre, Adrei Paskevich, and Claude Marché. Why3: Shepherd your herd of provers. In *Proceedings of the First International Workshop on Intermediate Verification Languages, Boogie*, 2011.

[BGTY11] Bernhard Beckert, Christoph Gladisch, Shmuel S. Tyszberowicz, and Amiram Yehudai. KeYGenU: combining verification-based and capture and replay techniques for regression unit testing. *Int. J. Systems Assurance Engineering and Management*, 2(2):97–113, 2011.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan 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.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[CGS09] Jean-François Couchot, Alain Giorgetti, and Nicolas Stouls. Graph Based Reduction of Program Verification Conditions. In Hassen Saïdi and N. Shankar, editors, *AFM'09, Automated Formal Methods (co-located with CAV'09)*, pages 40–47, Grenoble, France, 2009. ACM Press.

[Cha04] Patrice Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, 2004.

[Cha10] Patrice Chalin. Engineering a sound assertion semantics for the verifying compiler. *IEEE Trans. Software Eng.*, 36(2):275–287, 2010.

[CJRS13] Koen Claessen, Moa Johansson, Dan Rosen, and Nick Smallbone. HipSpec : Automating inductive proofs of program properties. In Jacques Fleuriot, Peter Höfner, Annabelle McIver, and Alan Smaill, editors, *ATx'12/WInG'12*, volume 17 of *EPiC Series*, pages 16–25. EasyChair, 2013.

[CKM12] Cyrille Comar, Johannes Kanig, and Yannick Moy. Integrating formal program verification with testing. In *Proc. Embedded Real Time Software and Systems*, Toulouse, February 2012.

[CSH10] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *Proceedings of the 4th International Conference on Tests and Proofs*, TAP'10, pages 6–21, Berlin, Heidelberg, 2010. Springer-Verlag.

[DCKP13] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In Pascal Fontaine and Amit Goel, editors, *SMT 2012*, volume 20 of *EPiC Series*, pages 22–31. EasyChair, 2013.

[DFM11] Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy. Correct code containing containers. In *5th International Conference on Tests & Proofs (TAP'11)*, Zurich, June 2011.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *LPAR Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[GKM11] Jérôme Guitton, Johannes Kanig, and Yannick Moy. Why Hi-Lite Ada? In *Proceedings of Boogie 2011, the 1st International Workshop on Intermediate Language Verification*, August 2011.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

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

[HKM13] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis verification competition 2012 – organizer's report. Technical Report 2013-01, Department of Informatics, Karlsruhe Institute of Technology, 2013. Available at `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000034373`.

[HLL+12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16, 2012.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[Hoa03] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50:2003, 2003.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[KSD12] Johannes Kanig, Edmond Schonberg, and Claire Dross. Hi-Lite: the convergence of compiler technology and program verification. In *Proceedings of the 2012 ACM conference on High integrity language technology*, HILT '12, pages 27–34, New York, NY, USA, 2012. ACM.

[KvLT+12] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In *Proceedings of the 6th international joint conference*

on Automated Reasoning, IJCAR'12, pages 378–392, Berlin, Heidelberg, 2012. Springer-Verlag.

[Lei05] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, March 2005.

[LMK13] David Lesens, Yannick Moy, and Johannes Kanig. Formal validation of aerospace software. In *Proceedings of the DASIA 2013 – DAta System In Aerospace – Conference*, 2013.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[O'N12] Ian O'Neill. SPARK – a language and tool-set for high-integrity software development. In Jean-Louis Boulanger, editor, *Industrial Use of Formal Methods: Formal Verification*. Wiley, 2012.

[RTC92] RTCA. DO-178B: Software considerations in airborne systems and equipment certification, 1992.

[RTC11] RTCA. DO-178C: Software considerations in airborne systems and equipment certification, 2011.

[SB12] Florian Schanda and Martin Brain. Using Answer Set Programming in the Development of Verified Software. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 72–85, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[TFNM11] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Usable Verification of Object-oriented Programs by Combining Static and Dynamic Techniques. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM'11, pages 382–398, Berlin, Heidelberg, 2011. Springer-Verlag.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.