Paper “Rail, Space, Security: Three Case Studies for SPARK 2014”

File name:	MFR.DOCX
Version:	1.0
Date:	2013-11-21
Author(s):	C. Dross, P. Efstathopoulos, D. Lesens, D. Mentré, Y. Moy

Classification Status:	Non confidential
Export Control Status:	NLR
Edition Status:	Approved

Approved by:	SPT	Date:	2013-11-21
Diffusion list:	Public		

Rail, Space, Security: Three Case Studies for SPARK 2014

Claire Dross[4], Pavlos Efstathopoulos[1], David Lesens[2], David Mentré[3] and Yannick Moy[4]
1: Altran UK Limited, 22 St Lawrence Street, Bath BA1 1AN (United Kingdom),
2: Astrium Space Transportation, 51-61 route de Verneuil F-78130 Les Mureaux (France),
3: Mitsubishi Electric R&D Centre Europe, 1 allée de Beaulieu, CS 10806, F-35708 Rennes (France),
4: AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France)

Abstract SPARK is a subset of the Ada programming language targeted at safety and security critical applications. SPARK 2014 is a major evolution of the SPARK language and toolset, that integrates formal program verification in the existing development and verification processes, in order to decrease the cost of verification for software subject to certification constraints. We present industrial case studies in three different certification domains that show the benefits of using formal verification with SPARK 2014.

Keywords System formal development, Verification and validation, Certification and dependability

1 Introduction

SPARK 2014 is a major evolution of the SPARK subset of Ada and associated formal verification toolset with two main objectives: (i) being accessible to non-expert users, and (ii) being compatible with testing.

In this paper, we describe how SPARK 2014 achieves these objectives, based on practical use of the language and associated formal verification tool GNATprove on three industrial case studies developed in the context of the Hi-Lite[1] and openETCS[2] research projects. The first case study by Mitsubishi Electric R&D Centre Europe evaluates the use of Ada and SPARK on a small subset of requirements for the European Train Control Systems. The second case study by Astrium Space Transportation is an extensive evaluation of the technology over a period of three years, consisting in the re-development of various parts of a Flight Control and Vehicle Management software. The third case study by Altran compares the benefits of the new technology w.r.t. previous version, on the well-known Tokeneer case study in SPARK.

2 SPARK 2014

SPARK is a subset of the Ada programming language targeted at safety and security critical applications. SPARK builds on the strengths of Ada for creating highly reliable and long-lived software. SPARK restrictions ensure that the behavior of a SPARK program is unambiguously defined, and simple enough that formal verification tools can perform an automatic diagnostic of conformance between a program specification and its implementation. The SPARK language and toolset for formal verification has been applied for many years in on-board aircraft systems, control systems, cryptographic systems, and rail systems [3, 12]. The new version SPARK 2014 builds on the new specification features added in Ada 2012 [2], so formal specifications are now understood by the usual development tools and can be executed.

2.1 Key Language Features

The most useful feature in SPARK 2014 is the ability to specify a contract on subprograms. Subprogram contracts were popularized in the Design-by-Contract approach [11] as a means to separate responsibilities in software between a caller and a callee. The callee's *precondition* states the respon-

[1] http://www.open-do.org/projects/hi-lite/

[2] Part of this work was funded by the DGCIS (Grant No. 112930309) in the context of the ITEA2 project openETCS http://openetcs.org/.

sibility of its caller, while the callee's *postcondition* states the responsibility of the callee itself. For example, the following contract for procedure Swap specifies that it should be called with index parameters within the range of the array parameter, and that Swap will ensure on return that the corresponding values in the array have been swapped. Attribute Old in the postcondition is used to refer to values on entry to the subprogram.

```
procedure Swap (A : in out Arr; X, Y : Idx) with
  Pre  ⇒ X in A'Range and Y in A'Range,
  Post ⇒ A(X) = A(Y)'Old and A(Y) = A(X)'Old;
```

New expressions make it easier to express contracts. If-expressions and case-expressions are the expression forms which correspond to the usual if-statements and case-statements. Note that 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, only on a bounded domain. 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 contracts.

The second most useful feature in SPARK 2014 (after contracts) is the ability to specify properties of loops. A loop invariant expresses the cumulated effect of the loop up to that point. For example, the following loop invariant expresses that the array A has been zeroed out up to the current loop index J, and that the rest of the array has not been modified. Attribute Loop_Entry is used to refer to values on entry to the loop.

```
pragma Loop_Invariant
  (for all K in A'Range ⇒
     (if K ≤ J then
        A(K) = 0
      else
        A(K) = A'Loop_Entry(K)));
```

2.2 Benefits of Executable Contracts

Traditionally, contracts have been interpreted quite differently depending on whether they were used for run-time assertion checking or for formal program verification. For run-time assertion checking, contracts have been interpreted as assertions on entry and exit of subprograms. For formal program verification, assertions have typically been interpreted as formulas in classical first-order logic. This was the situation with SPARK prior to SPARK 2014. Practitioners have struggled with this interpretation, which was not consistent with the run-time assertion checking semantics [6].

SPARK 2014 reconciles the logic semantics and executable semantics of contracts, so users can now execute contracts, debug them like code, and test them when formal verification is too difficult to achieve. Furthermore, by keeping the annotation language the same as the programming language, users don't have to learn one more language.

All the previously presented contracts and assertion pragmas lead to run-time assertions. If a given property is not satisfied at run time, an exception is raised with a message indicating the failing property, for example on the procedure Swap:

```
failed precondition from swap.ads:4
```

Another key benefit of executable contracts is that they can be used by other tools working at the level of code. For example, the CodePeer[3] static analysis tool uses contracts and assertion pragmas to issue more precise messages. Most notably, this allows also combining the results of formal verification and testing, when only part of a program is formally analyzed [7].

2.3 Key Tool Features

GNATprove is the formal verification tool that analyzes SPARK 2014 code. It performs two different analyses: (i) flow analysis of the program and (ii) proof of program properties.

Flow analysis checks correct access to data in the program: correct access to global variables and correct access to initialized data. It is a fast static analysis (analysis time typically comparable with compilation time).

Proof is used to demonstrate that the program is free from run-time errors, and that the specified contracts are correctly implemented. It internally generates mathematical formulas for each property, that are given to the automatic prover Alt-Ergo[4]. If Alt-Ergo manages to prove the formula in the

[3] http://www.adacore.com/codepeer

[4] http://alt-ergo.ocamlpro.com/

given time, then the property is known to hold. Otherwise, more work is required from the user to understand why the property is not proved.

As proof requires interactions between the user and the tool until the specification can be proved automatically, the efficiency and the granularity at which the tool can be applied are critical. For efficiency, GNATprove uses a compilation-like model, where only parts that are impacted by a change need to be reanalyzed, and a fast generation of formulas [9]. For convenient interaction, GNATprove allows focusing on a single unit, a single subprogram inside this unit, or even a single line inside this subprogram.

A very useful feature of GNATprove to investigate unproved properties is its ability to display the paths in the program that lead to unproved properties. This path can be displayed in GPS[5] or in Eclipse[6], the two Integrated Development Environments which support SPARK 2014. The user can also change the parameters of the tool to perform more precise proofs, at the expense of longer analysis time.

3 Train Control Systems

The openETCS European project aims at making an open-sourced, open-proofs reference model of ETCS (European Train Control System). ETCS is a radio-based train control system aiming at unifying train signaling and control over all European countries. Organized in several levels, ETCS can range from, at Level 0, a simple Automatic Train Protection system monitoring train speed to, at Level 3, a fully featured radio-based train control system where trains inform a Radio Block Centre about their location and receive Movement Authorities, using cab signaling instead of track-side signaling.

We made some experiments with SPARK 2014 to see if one could formalize the ETCS System Requirement Specification (SRS, "*ERA UNISIG SUBSET-026*"). We should acknowledge that using SPARK 2014 for formalizing *system* requirements is a bit excessive and out of scope for the language. We made nonetheless this formalization attempt because we wanted to give a formal semantics to this system specification and some of the content of the ETCS specifications is quite low-level.

We made several experiments but due to space constraints we will only detail one of them. The complete code is available online[7]. This example is the coding of step functions, or piecewise constant functions, used in the *Speed and distance monitoring* section (SRS §3.13). Such functions are used to model for example speed restrictions along distance. One of our main goals is to model the merge of two speed restrictions, taking at each point the most restrictive (*i.e.* smaller) one.

To encode step functions, we used the following data structure:

```
type Num_Delimiters_Range is range 0 .. 10;
type Function_Range is new Natural;
type Delimiter_Entry is record
   Delimiter : Function_Range;
   Value     : Float;
end record;
type Delimiter_Values is array
  (Num_Delimiters_Range) of Delimiter_Entry;
type Step_Function_t is record
   Num_Delim : Num_Delimiters_Range;
   Step      : Delimiter_Values;
end record;
```

A step function of type `Step_Function_t` can have up to 11 steps separated by 10 delimiters, stored together with the initial value of the step function (as first element of the array) in component `Step`. Each delimiter of type `Delimiter_Entry` contains the delimiter position (`Delimiter`) and the associated constant value (`Value`) for the function. The number of delimiters used is stored in component `Num_Delim`. We defined several subprograms that query or update step functions.

3.1 Formalization of Properties

We wanted to check full functional correctness of this critical unit. We used contracts to express the specifications of all subprograms.

For example, we specified that, given a valid (*i.e.* with strictly increasing delimiters) step function `SFun` and a point `X`, function `Minimum_Until_Point` returns a value of the step function (2) where the step function reaches its minimum on the given domain (1).

[5] http://www.adacore.com/gnatpro/toolsuite/gps/

[6] http://www.adacore.com/gnatpro/toolsuite/gnatbench/

[7] https://github.com/openETCS/model-evaluation/tree/master/model/GNATprove-MERCE

```
function Minimum_Until_Point
    (SFun : Step_Function_t; X : Function_Range)
                                  return Float
with
Pre  ⇒ Is_Valid(SFun),
Post ⇒
  -- (1) Returned value is the minimum before X
  (for all i in
     Num_Delimiters_Range'First..SFun.Num_delim ⇒
      (if X ≥ SFun.Step(i).Delimiter then
        Minimum_Until_Point'Result
           ≤ SFun.Step(i).Value))
  and
  -- (2) Returned value is a value of the step
  --       function before X
  ((for some i in
      Num_Delimiters_Range'First..SFun.Num_delim ⇒
       (X ≥ SFun.Step(i).Delimiter
         and
       (Minimum_Until_Point'Result
         = SFun.Step(i).Value))));
```

The most complex subprogram we specified is `Restrictive_Merge`. We specified that, given two valid step functions `SFun1` and `SFun2` without too many delimiters, `Restrictive_Merge` generates a step function `Merge` which is valid and contains all the delimiters of `SFun1` and `SFun2` and also, for each of those delimiters, the value of the `Merge` step function is the minimum over both input step functions.

Overall, the number of lines for contracts and assertion pragmas (in particular loop invariants) is about the same as the number of lines of code. We could have simplified the contracts by specifying that `Is_Valid` is a type invariant of `Step_Function_t`, but this Ada feature is not yet supported in SPARK 2014, although it is planned for the future.

3.2 Formal Verification Results

The first goal of this experiment was to check if SPARK 2014 was expressive enough to describe the objects of the requirements: requirement text, transition tables, breaking curve equations, *etc.* Overall, we were quite satisfied, as we were able to express most of the requirements in a formal way. The very expressive data structures of SPARK 2014 (records, arrays, enumerations, *etc.*) were very helpful compared to other specification languages like B Method [1] (lacking usable record structures) or ACSL [4] for C programs (lacking record with variant part or bounded integers). We found that it lead to quite readable specifications.

The second goal was to evaluate the automatic proving capabilities of SPARK 2014 on some parts of the specification. We were able to prove the complete absence of run-time errors, plus the functional contracts of all subprograms except the one of `Restrictive_Merge`. In this procedure, the postcondition and the loop invariant (added to be able to prove the postcondition) could not be automatically proved by Alt-Ergo. The main reason is that the proof context is too big and Alt-Ergo gets lost in all the possible quantifier instantiations. We have checked that some parts of the loop invariant could be automatically proved if the proof context was manually pruned of irrelevant hypotheses. Moreover, we compiled and tested the contracts and assertion pragmas, which increased our confidence in their correctness.

3.3 Lessons Learned

We were rather pleased by the expression capabilities of SPARK 2014, making specification and code writing rather easy and, more importantly, clearer for the reader. The ability inherited from Ada to define new data types for specific ranges, which are incompatible with other types, is crucial in this regard.

Another important finding is that the code should be written with proof in mind. For example, function `Minimum_Until_Point` was initially unprovable, because an early exit in a loop lead to the formula

$$(\forall\ K.A(K-1) < A(K)) \wedge X < A(1) \\ \rightarrow (\forall\ K.K > 1 \rightarrow X < A(K))$$

which requires support for induction in the automatic prover, a feature still missing from most automatic provers. As the early exit was not necessary for correct behavior, it was sufficient to remove it to achieve automatic proof with Alt-Ergo.

A third finding is that contracts that can be automatically proved are not always the most natural contracts, *i.e.* contracts a reviewer would understand more easily. For example, for `Restrictive_Merge` procedure, we would have preferred to write that the resulting function is the minimum of both input functions for all possible input values:

```
Post ⇒
  (for all i in Function_Range ⇒
   (Get_Value(Merge, i) =
  Min(Get_Value(SFun1, i), Get_Value(SFun2, i))));
```

It would have been impossible to prove this contract automatically, so we wrote instead that the resulting function is the minimum of both input functions for all possible delimiter values, which is logically equivalent and proved automatically:

```
Post =>
  (for all i in
     Num_Delimiters_Range'First .. Merge.Num_Delim =>
    (Merge.Step(i).Value = Min
      (Get_Value(SFun1, Merge.Step(i).Delimiter),
       Get_Value(SFun2, Merge.Step(i).Delimiter))));
```

Ideally, the tool should allow users to manually prove contracts that cannot be proved automatically, like done for example in B Method tools.

Our last finding, not entirely surprising, is that writing the correct loop invariant for a complex loop is not an easy task, as we experienced it for procedure `Restrictive_Merge`. It can require several hours of work of skilled people, trained in the proof environment. In such cases, the ability to compile and test the loop invariant is very useful to help "debug" the loop invariant.

Overall, we think that the programmer should be trained in the proof environment, much like a programmer exploits feedback of debuggers and tests to debug her program. Moreover, as a complete proof of the program can be very costly, a proof methodology must be defined to avoid spending too much time on proofs that could be broken by future code changes.

4 Flight Control and Vehicle Management in Space

This case study was carried out as part of project Hi-Lite as a means to evaluate the adequacy of formal verification for future space programs. Detailed results have been published in [10].

A typical space flight program is made up of two parts, both of which are considered in our case study: Flight control (or more generally numerical command and control algorithm) and Mission and Vehicle Management.

4.1 Numerical Command and Control Algorithms

Numerical command and control algorithms take as input floating point values, perform some numerical computations (using the classical basic mathematical operators such as additions, subtractions, multiplications, divisions, absolute values, trigonometry or operations on vectors and arrays, *etc.*) and return floating point results. Such algorithms generally have a retro-action loop, *i.e.* internal states.

It is generally not possible to define interesting functional contracts for such code. Indeed, the functional contract of the equation "X := A * Y + Cos (Z)" is just itself (*i.e.* it is not possible to specify in a more abstract way this equation). Then, instead of defining functional contracts, the objective on this kind of software is the proof of absence of run-time errors (such as division by zero) and the correctness of variable ranges (such as, for instance, a velocity shall always be between 0 and 25 km/s).

SPARK 2014 has been first experimented on a solar wing management software (for a spacecraft such as the ATV – Automated Transfer Vehicle). This piece of code uses a mathematical library whose implementation is not in SPARK 2014, implying that it could not be formally proved, but only tested. However, the interface of this mathematical library is in SPARK 2014. The contracts defined on the mathematical library can then be used to prove the application's code. For example:

```
function Sin (X : T_Float) return T_Float
with
  Pre  => X in -C_2Pi .. C_2Pi,
  Post => Sin'Result in -1.0 .. 1.0;
```

4.2 Mission and Vehicle Management

The Mission and Vehicle Management of a spacecraft is described by the ECSS (European Cooperation for Space Standardization) standard *ECSS-E-ST-70-01C "Space engineering – Spacecraft on-board control procedures"*. This standard defines the general principles of an On Board Control Procedure (OBCP). An OBCP is in practice represented by a simplified programming language interpreted on-board the spacecraft. This interpreter is generally at the highest level of criticality of the spacecraft. The implementation of this interpreter in SPARK 2014 is table driven and relies greatly on rich features of Ada such as generic packages (allowing an easy customization of the code) and discriminants (ensuring a strict typing of the code, even in case of heterogeneous communication between components of the system).

4.3 Formalization of Properties

The contracts defined on algorithmic code are mainly related to the ranges of variables.

The contracts defined on mission and vehicle management code have (among others) the objectives of ensuring the permanent consistency of the software tables, the permanent consistency between the Mission and Vehicle Management function and other functionalities (such as, for instance, the solar wing management), some functional properties such as mutual exclusion between executions of some automated procedures are respected and the absence of run-time errors.

A majority of the program could be formally analyzed (1505 out of 2054 subprograms), although some subprograms fell outside the boundary of SPARK 2014. Subprograms that could not be analyzed either used access types (a.k.a. “pointers”) that are outside the language (87 subprograms), or unchecked type conversions that are not verified (377 subprograms). The 377 subprograms using unchecked type conversions are very small subprograms defined in a library for reading external inputs and could be validated by intensive testing. The remaining unanalyzed subprograms used features that are expected to be included in a future version of the language: class wide types (53 subprograms), tagged types (81 subprograms) and specific attributes (10 subprograms). These correspond to Object Oriented Programming features, allowing dynamic binding of subprograms depending on the type of objects at run time.

Component	**Size**	**Pre**	**Post**	**Time**
Library	694	34	39	233
Algorithms	795	8	2	653
Time	13	0	2	10
Variable	260	41	30	118
Variables	438	43	31	274
Events	249	16	17	371
Expressions	1253	93	79	1992
Parameters	75	0	1	279
Units	463	13	8	2921
Sequences	276	5	5	7803
OBCP	714	33	15	13705

Subprograms that were analyzed were first specified with a precondition and a postcondition. The above table provides for each component: the number of lines of code (“Size”), the number of preconditions (“Pre”), the number of postconditions (“Post”) and analysis time in seconds (“Time”).

4.4 Formal Verification Results

All the contracts have been checked by dynamic testing. This phase is quite classical, except for the fact that preconditions and postconditions were also tested. Then, GNATprove has been applied. The following table provides for each type of checks, their number and the number proved:

Features	**Checks**	**Proved**	**%**
division check	22	22	100
overflow check	164	164	100
precondition	1410	1400	99
postcondition	369	344	93
range check	232	194	87
assertion	967	961	99
index check	184	46	25
discriminant check	2334	2327	99
loop initialization	14	12	86
loop preservation	14	14	100

These results are globally satisfactory, a quite limited number of checks having not been proved. All unproved checks were analyzed to determine why they were not proved, which uncovered a few limitations of the tool: some algorithmic functions (*e.g.* trigonometric functions) are not completely known to GNATprove and therefore cannot be used in proofs, GNATprove had some difficulties to take into account rounding, GNATprove could not prove non linear equations involving floating-point values, GNATprove could not prove some index checks in discriminated record. The remaining non proved checks were due to too complex subprograms. These subprograms shall be split in several smaller subprograms to be proved.

4.5 Lessons Learned

A precise process was followed for the development of this case study: (1) writing of the contracts of each subprogram, (2) development of the body and of the tests, (3) test of the software with executable contracts, and potentially fixes, (4) formal proof of contracts. The errors detected in the testing phase were in the code in 50% of the cases, and in the contracts in the remaining 50%.

The first run of GNATprove was often not sufficient to achieve 100% automatic proof. Investigating the failed proofs was sometimes very difficult.

While errors in the code are generally quite easy to find (as it is the classical activity of an engineer), missing contracts or assertion pragmas were almost always very difficult to detect without the help of an expert in SPARK 2014. Moreover, conclusions on a tool limitation shall almost always be confirmed by a senior expert on SPARK 2014.

The quality of the code has been dramatically improved thanks to both executable contracts and formal proof. The writing of contracts for testing should become a natural activity of any software developer. Understanding why a proof (which may seem sometimes really obvious) does not work may require the involvement of a SPARK 2014 expert. The writing of effective contracts for formal proof requires a dedicated training and sometimes the involvement of an expert in SPARK 2014.

In order to efficiently use GNATprove in an industrial context, we think following points should be improved: ability to prove generic units once (instead of proving each instance like done currently), need to use a sound model of floating-points instead of real numbers and automatic generation of some loop invariants.

5 Biometric Access to a Secure Enclave

Tokeneer[8] is a highly secure biometric software system that was originally developed by Altran. The system provides protection to secure information held on a network of workstations situated in a physically secure enclave. The Tokeneer project was commissioned by the US National Security Agency (NSA) to demonstrate the feasibility of developing systems to the level of rigor required by the higher assurance levels of the Common Criteria. The requirements of the system were captured using the Z notation and the implementation was in SPARK 2005. The original development artifacts, including all source code, are publicly available.

During this study, the source code for Tokeneer has been translated into SPARK 2014. The core system now consists of approximately 10,000 lines of SPARK 2014 code. There are also approximately 3,700 lines of supporting code written in Ada which mimick the drivers to peripherals connected to the core system.

5.1 Converting SPARK 2005 to SPARK 2014

For the majority of the code, translating SPARK 2005 to SPARK 2014 was very straightforward since most of the original annotations map directly to the new ones. This section will focus on the more interesting occasions where the conversion was non-trivial.

Often, it is convenient to introduce a function which exists solely for the sake of proof and does not contribute at all in the final executing program. The constructs which achieve this functionality for SPARK 2005 and SPARK 2014 respectively are proof functions and ghost functions. In SPARK 2005, the behavior of a proof function was defined with user rules, which are axioms expressed in a special syntax and given to the proof system. In SPARK 2014, this behavior is simply given by the body of the ghost function.

The Global aspect has an additional mode `Proof_In` in SPARK 2014 w.r.t. SPARK 2005. This mode is used to specify global variables of a subprogram which appear solely within contracts and assertions. Therefore, the translation required separating global variables of mode `in` that were used exclusively inside annotations (translated as `Proof_In` global variables), from those that were used in code (translated as `Input` global variables).

Parts of the original code were specially annotated to be ignored during formal verification, because they contained constructs which were outside the Ada subset supported by SPARK 2005. As SPARK 2014 supports a bigger subset of Ada than SPARK 2005, a lot of this code is now formally analyzable. The two constructs most often encountered, which fall under this category, are string concatenation and array slices.

The information that values of variables respect the constraints of their type is available for proofs with the new toolset. This simplifies both the work of the programmer and the code itself. In SPARK 2005, it was necessary to add many lines of assertions to repeat that the values of variables and components are within the bounds allowed by their type, as this information was lost inside loops. The new tools are more sophisticated and preserve more of the context.

[8] www.adacore.com/sparkpro/tokeneer

5.2 Formalization of Properties

The contracts that have been added in the Tokeneer source code consist of information flow contracts expressed as Depends aspects and functional behavioral contracts expressed as Pre and Post aspects.

Aspect/Pragma	Num. of occurrences
Global	197
Refined_Global	71
Depends	202
Refined_Depends	40
Pre	28
Post	41
Assume	3
Loop_Invariant	10

The Depends aspects (not presented previously for lack of space) facilitate flow analysis of the code. Flow analysis detects improper initialization, identifies ineffective assignments and ensures secure flow of information.

The Pre and Post aspects enable the prover to show that the code is free from run-time exceptions, such as buffer overflow or divide-by-zero and assist in proving that key security properties are guaranteed by the implementation.

In order to measure the expressiveness and proving power of the SPARK 2014 tools, a specific package, on which functional behavior proof had not been attempted, was selected and then fully augmented with functional behavior annotations: package `admin`. This package contains the state of the administrator of the system and a set of operations that administrators can perform.

To illustrate how functional behavior contracts were added, we will consider subroutine `OpIsAvailable` of package `admin`. An administrator can be either a `UserOnly`, a `Guard`, an `AuditManager` or a `SecurityOfficer`. Each type of administrator has a set of predefined operations that it is allowed to perform. Function `OpIsAvailable` takes as input an administrator and a string that is read from the keyboard and determines if this string corresponds to an operation available to the administrator. If the operation is indeed available, then this operation is returned, otherwise `NullOp` is returned.

The SPARK 2005 postcondition served only as a test case. It ensured that if a non null operation was returned and if that operation was `OverrideLock` then the administrator was of type `Guard`. This annotation was incomplete since it did not specify any other kind of valid combinations of administrator types and their corresponding operations or under which circumstances `NullOp` should be returned. The SPARK 2005 tools proved all properties associated with the non-augmented `admin` package but a total of 12 user rules had to be provided. The effort associated with proving the completed version of this postcondition would have been significant and hence this was not attempted.

```
function OpIsAvailable
  (TheAdmin : T; KeyedOp : Keyboard.DataT)
  return OpAndNullT;
with
  Pre  => IsPresent(TheAdmin),
  Post => (for some Op in Opt =>
             Str_Comp(KeyedOp, Op)
               and AllowedOp(TheAdmin, Op)
               and OpIsAvailable'Result = Op)
          xor OpIsAvailable'Result = NullOp;
```

This SPARK 2014 Post aspect states that we have two mutually exclusive cases. If there exists some operation in `Opt` that matches the string read from the keyboard and this operation is allowed for the current administrator then the result of function `OpIsAvailable` is this operation, otherwise, the result is `NullOp`.

5.3 Formal Verification Results

The original source code of Tokeneer was proven to be free of run-time exceptions and some key security properties were proven to hold but full functional proof was not performed on the entirety of the code.

The SPARK 2014 tools discharge all 24 verification conditions associated with the augmented `admin` package in a matter of seconds.

More than 95% of the checks that were associated with the converted code were automatically discharged. The remaining 5% consisted of checks that either derived from code that was previously not analyzable (and hence no provision was in place to assist in their provability) or required something similar to the SPARK 2005 user rules to assist the prover in discharging them (this can potentially also be achieved through utilizing a combination of pragmas Assert and Assume). The typical cases that require additional assumptions are those on which combinations of several non-trivial mathematical transformations would have to be applied when performing a manual proof.

5.4 Lessons Learned

Thanks to the new toolset's proving power, users are no longer required to add intermediate cutpoints. In the Tokeneer case study, pragma `Assert` did not have to be provided to facilitate proof. However, it is hypothesised that its usage could improve readability of the code since it would signify that a certain property holds at a given point in the code.

While augmenting package `admin`, it was noticed that formulating the functional behavior annotations based on the low level Z specifications was very intuitive. This suggests that it might be possible for future projects to skip this step and directly provide SPARK 2014 annotated specifications.

An interesting case of how executable semantics affect the way code and annotations have to be written was uncovered while attempting to prove package `AuditLog`. Converting SPARK 2005 annotations into their SPARK 2014 equivalents introduced several run-time checks that previously did not exist. The extra checks are a byproduct of the contracts and assertions being executable (Section 2.2). Some of these checks were not automatically discharged. A more in-depth investigation revealed that an invariant was missing from the original specifications. This invariant described the property of always using at least one log file (`UsedLogFile.Length >= 1`)[9].

When analyzing a single package in isolation, it was easy to understand which verification conditions were proved and which were not. However, when more than one file was analyzed in a single run, due to the magnitude of the output, it became increasing complicated to retain supervision of both individual and overall provability. Having a tool that summarizes the results and informs about remaining undischarged checks would greatly assist. For instance, this would enable users to focus their efforts only on proving packages for which 100% automated proof was achievable and resort to testing for the rest.

6 Common Findings and Dissimilarities

Not surprisingly, the lessons learned in all three case studies partly reflect the level of expertise in Ada and SPARK of the engineers involved: first-time experience with Ada and SPARK for the first case study, extensive experience in previous versions of Ada and SPARK for the second case study, member of the team developing GNATprove for the third case study.

The three case studies confirm some good properties of SPARK 2014. One of the principal objectives of SPARK 2014 is to offer an expressive language for formal verification. As stated by Jonathan P. Bowen in Ten Commandments of Formal Methods [5], *Thou shalt choose an appropriate notation.* The three case studies confirm that expressiveness is indeed perceived as an advantage of SPARK 2014. It provides both rich data-structures such as records and enumerations (Section 3) and support for advanced coding structures such as generic packages and discriminants (Section 4). It is close enough to SPARK 2005 to allow an easy translation of annotations but also supports constructs that were previously excluded from the language (Section 5).

Tests remain an important means of gaining confidence in a program's specification and implementation, even when formal methods are involved. As stated in Bowen's ninth commandment of formal methods, *Thou shalt test, test, and test again.* Executable contracts are a key language feature (see Section 2.2) of SPARK 2014 as it allows to execute the contracts while testing. They were found to be beneficial both to increase confidence in annotations (Section 3) and to improve overall code quality (Section 4). What is more, the constraint implied by executability of contracts does not seem to be too important as additional checks introduced by the verification of absence of run-time errors in contracts on a project that had not be annotated with this constraint in mind could be discharged (Section 5).

Another objective of SPARK 2014 was to improve the amount of proofs that could go through automatically with respect to SPARK 2005. The case study in Section 5 shows that obtaining automated proofs with SPARK 2014 requires smaller

[9] www.open-do.org/wp-content/uploads/2013/05/Industrial_Case_Studies_Final_Report.pdf

loop-invariant, as information about bounds of variables is preserved, less user rules, as ghost functions can now have bodies, and less user written additional assertions.

The case studies also raise some usability issues. SPARK 2014 strives to define a subset of Ada as big as possible while remaining amenable to formal verification. For example, pointers are excluded but a library of containers adapted to formal specification is provided to alleviate this restriction [8]. The case studies show that both code and contracts must also be adapted for formal proof to go through automatically (Sections 3 and 4).

Determining why a proof doesn't succeed is a really difficult task. Even if specific feedback can be provided by the proof environment for a failed proof in the form of an execution trace, the results of the case-study show that there is room for improvement in that matter. For example, the tool could provide inputs on which the annotation is not verified. What is more, it appears that programmers should be trained to debug their proof and annotations (Sections 3 and 4).

Finally, as stated in Bowen's fourth commandment, *Thou shalt have a formal methods guru on call*. If automated formal verification of run-time errors is achievable by an implementer, the case studies show that, as the desired properties become more complex, their expression and their verification may require the involvement of an expert (Sections 3 and 4).

7 Conclusion

Railway, space, and security are three domains subject to an important certification requirement that implies costly verification processes. The three case studies show that verification with SPARK 2014 can bring a solution to this problem thanks to an expressive language, executable contracts, and improved provability. Even if there is still room for improvement in usability, the current toolset already allows integration of formal proofs into the standard developer's workflow.

References

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[2] J. Barnes. Ada 2012 Rationale. 2012.

[3] J. Barnes. *SPARK: The Proven Approach to High Integrity Software.* Altran Praxis, 2012.

[4] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language.* http://frama-c.com/download/acsl.pdf.

[5] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.

[6] P. Chalin. Engineering a Sound Assertion Semantics for the Verifying Compiler. *IEEE Trans. Software Eng.*, 36(2):275–287, 2010.

[7] C. Comar, J. Kanig, and Y. Moy. Integrating Formal Program Verification with Testing. In *Proc. ERTS*, 2012.

[8] C. Dross, J.-C. Filliâtre, and Y. Moy. Correct code containing containers. In *Tests and Proofs*, pages 102–118. Springer, 2011.

[9] K. R. M. Leino. Efficient Weakest Preconditions. *Inf. Process. Lett.*, 93(6):281–288, March 2005.

[10] D. Lesens, Y. Moy, and J. Kanig. Formal validation of aerospace software. In *Proc. DASIA'11*, May 2013.

[11] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., 1st edition, 1988.

[12] I. O'Neill. SPARK – a language and tool-set for high-integrity software development. In *Industrial Use of Formal Methods: Formal Verification.* Wiley, 2012.