Climbing the Software Assurance Ladder - Practical Formal Verification for Reliable Software

* Claire Dross[1], Guillaume Foliard[2], Théo Jouanny[3], Lionel Matias[2], Stuart Matthews[4], Jean-Marc Mota[5], Yannick Moy[1], Pascal Pignard[3], and Romain Soulat[5]

[1] AdaCore, F-75009 Paris
[2] Thales Air Systems, F-91470 Limours
[3] Thales Communications & Security, F-49300 Cholet
[4] Altran, Bath BA1 1AN, United Kingdom
[5] Thales Research & Technologies, F-91767 Palaiseau

Abstract. There is a strong link between software quality and software reliability. By decreasing the probability of imperfection in the software, we can augment its reliability guarantees. At one extreme, software with one unknown bug is not reliable. At the other extreme, perfect software is fully reliable. Formal verification with SPARK has been used for years to get as close as possible to zero-defect software. We present the well-established processes surrounding the use of SPARK at Altran UK, as well as the deployment experiments performed at Thales to fine-tune the gradual insertion of formal verification techniques in existing processes. Experience of both long-term and new users helped us define adoption and usage guidelines for SPARK based on five levels of increasing assurance that map well with industrial needs in practice.

1 Introduction

Taken literally, reliable software is the notion that we can rely on software to perform as intended. This is also how the international standard bodies and academic experts define it, as phrased in IEC 60050 terms applied to software: *"reliability [is the] ability to perform as required, without failure, for a given time interval, under given conditions"*. Currently, almost no software is reliable in this sense, because the intention is usually expressed in ambiguous natural language, and the confidence that software behaves as intended is obtained by a combination of development discipline (to avoid introducing errors) and partial testing of all the possible software behaviors (to detect errors that were introduced). Hence, reliable software today is more an aspiration when building the software than a quality of the software produced. However, a link between software quality and reliability does exist, and it was clarified by researcher John Rushby [25]:

* Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, https://www.adacore.com/proofinuse)

"probability of (im)perfection [..] provides a bridge between correctness, which is the goal of software verification (and especially formal verification), and the probabilistic properties such as reliability that are the targets for system level assurance."

This interpretation of reliable software as probably perfect software has been taken seriously in some companies like Altran UK, where specifications are routinely expressed with the precision of a formal language, and confidence is obtained by a combination of classical techniques plus the guarantees provided by the use of formal verification. Tools for formal verification of software have reached a degree of automation and usability that makes them suitable for use in commercial contexts across a large set of industries, from the well-established - space, railway, aerospace & defense - to industries that more recently included software as a critical component like automotive and medical. The main tool used at Altran UK for formal specification, programming and formal verification of software is SPARK [23], a subset of the Ada programming language targeted at safety- and security-critical applications. The use of SPARK allows Altran UK to provide assurance that software will not crash or behave erratically, and that critical properties are satisfied, which it demonstrates by committing to these properties with its customers.

While the benefits obtained by formal verification at Altran UK are clearly desirable, it may be intimidating for companies without formal verification know-how to start on this path. Knowing that others have replicated these benefits in other contexts is an important argument to make. Here, we are describing the experiments done at Thales with the support of AdaCore, during the years 2016 and 2017, to assess the costs and benefits of using formal verification of software using SPARK. With little investment in training (2 days in one experiment, self-training only in the other) and consulting (20 days in one experiment, online support only in the other), either provided externally or through self-training, the operational teams managed to specify intended behavior formally. The engineers in these teams were knowledgeable about Ada but not familiar with formal methods. The teams also proved critical properties of their software. Specifically: in multiple case studies the code was fully proved to be free of run-time errors (like buffer overflows and divisions by zero); in a subset of these case studies the code was proved to implement functional API specifications; while in another case study the code was proved to follow a specified safety automaton.

In addition, the collaboration of AdaCore and Thales resulted in a set of guidelines [1] that should be followed for an easier adoption of formal verification in existing projects, codebases and processes. These guidelines are based on five levels of assurance that can be achieved on software, in increasing order of costs and benefits. These guidelines are a major result of this work, as there is very little available guidance on the use of specific formal methods and tools in industrial context. It could be used as an inspiration by other formal verification platforms for software.

In section 2, we introduce the SPARK formal verification platform. In section 3, we present the experience of Altran UK, a long-time user of formal ver-

ification with SPARK, and how it relates to traditional assurance levels (DAL and SIL) considered in industry. In section 4, we present two studies carried at Thales, a recent user of formal verification with SPARK, and how such adoption can be facilitated by the use of suitable guidelines. We finish with related works in section 5 and conclude.

2 SPARK: Formal Verification Focused on Practical Use

For his PhD defense in 1969 on "A Program Verifier", J. King submitted a manuscript that started with these sentences [15]:

> This research is a first step toward developing a "verifying compiler". Such a compiler, as well as doing the standard translation of a program to machine executable form, attempts to prove that the program is "correct". In order to do this a program must be annotated with propositions in a mathematical notation which define the "correct" relations among the program variables. The verifying compiler then checks for consistency between the program and its propositions.

To most programmers, this may sound like a naive dream, whose illusory nature is exemplified by C.A.R. Hoare's call in 2003 for researchers to tackle the "verifying compiler" as a Grand Challenge, more than thirty years later. Yet, "verifying compilers" are available today. For example, the formal development environments Coq and Isabelle/HOL have been used to create a compiler for C [22] and a microkernel [19] which are guaranteed to be "correct" (related to a set of requirements and assumptions).

The problem is that these "verifying compilers" are operating on proof languages that are reserved for experts. Since King's PhD defense, there have been numerous attempts at defining practical "verifying compilers" for programming languages used in industry (first Pascal, then Ada, more recently Java and C#), none of which has succeeded in gaining industrial adoption. It is difficult to prove that a program is "correct", and it will remain so for the foreseeable future. As Rustan Leino, a prominent researcher in formal program verification, put it in 2010: *"Program verification is unusable. But perhaps not useless."* [21]

Departing from this academic tradition, SPARK has been focused on practical formal verification from the start. SPARK has been adopted in numerous large industrial projects and only critical parts of the software were proved "correct" with respect to full functional (*i.e.* behavioral) specification. SPARK was used to prove specific properties of interest about the software, like the absence of all possible run-time errors (no division by zero, no buffer overflow, etc.) and some user-specified safety or security properties.

SPARK is both a language and a toolset, supported by specific development and verification processes. In this article, we are focusing on the latest generation of SPARK technology, called SPARK 2014 [10], in which the specification language and the programming language have been unified as a subset of the programming language Ada 2012. Constraints on both program data and control

can be specified using respectively type contracts (predicates and invariants) and function contracts (preconditions and postconditions).

The concept of program contracts was invented by the researcher C.A.R. Hoare in 1969 in the context of reasoning about programs. In the mid-1980s, another researcher, Bertrand Meyer, introduced the modern function contract and type invariant in the Eiffel programming language [24]. In its simplest formulation, a function contract consists of two Boolean expressions: a precondition to specify input constraints and a postcondition to specify output constraints. Function contracts have subsequently been included in many other languages, either as part of the language (*e.g.* contracts for SPARK), as part of the standard library (*e.g.* CodeContracts for .NET [12]) or as an annotation language (such as JML for Java [5] or ACSL for C [2]). Type invariants may come in two forms, depending on whether they can be temporarily violated (type invariants in SPARK) or not (type predicates in SPARK). Contracts can be executed as runtime assertions, interpreted as logic formulas by analysis tools, or both.

The latest version of SPARK has opted for both. This design choice has far-reaching consequences. First, specifying properties of programs is similar to programming: there is no additional language to learn and the tools available to the programmer also work on specifications. Second, contracts are executable, which means that they can be tested and debugged like code. Another important design choice was to allow SPARK and Ada code to coexist in the same files. Hybrid verification is obtained by using proof on SPARK code and test on Ada code. This is possible because contracts can be executed, and because test and proof use the exact same semantics for contracts [4]. Other formal program verification technologies like Frama-C for C programs have made similar although not identical choices [20].

SPARK toolset focuses on automation and usability. Generation of implicit specifications lowers the cost of writing specifications, and generation of loop invariants, use of multiple state-of-the-art automatic provers, possible generation of counterexamples when proof fails, combination of static analysis and proof, all lower the cost of proof by reducing the time and effort required to prove that the code respects its contracts. Usability is similar to other tools in the developer's toolbox, mostly because formal verification can be performed by developers while they are developing, using their personal computers, thanks to the modularity and parallelization of the analysis.

We identify five levels of assurance that can be achieved with SPARK, which are - in increasing order:

1. Stone level - valid SPARK
2. Bronze level - initialization and correct data flow
3. Silver level - absence of run-time errors (AoRTE)
4. Gold level - proof of key properties
5. Platinum level - full functional correctness

At Stone level, strict SPARK rules are enforced on the code, having the effect of ensuring that a strong semantic coding standard is followed, which

leads to better code quality and maintainability. At Bronze level, the SPARK code is guaranteed to be free from a number of defects like reads of uninitialized variables. At Silver level, the SPARK code is guaranteed to be free of run-time errors. At Gold level, the SPARK code is guaranteed to respect key integrity properties. At Platinum level, the SPARK code is guaranteed to implement a complete specification of intended behavior. Note that each level builds on the previous one, so that at Platinum level the guarantees given by all the lower levels are also achieved.

3 The Practice of Formal Verification

Altran UK has a special relationship with the SPARK technology, being the heir of both PVL and then Praxis, the companies which have developed SPARK since 1987 [8]. Along the years, Altran has used SPARK both directly and in partnership with our customers - through training, support and consulting - in a number of project domains which range across air traffic management, airborne systems, avionics, railway control & protection, security and defense systems.

SPARK is used at Altran as an efficient means to both get as close as possible to zero-defect software and as a means to address the objectives of the relevant standards. This technical strategy has been subject to careful evaluation of costs and benefits, in order to apply formal verification where it brings more value to the business. At Altran UK, SPARK fits within an overall software development philosophy known as Correctness By Construction [6]. The key principles of this approach are:

- to use techniques that prevent the introduction of errors (e.g. language subsets);
- to maximize the ability to detect defects early (e.g. through the use of formal techniques);
- to generate assurance evidence as you progress.

The detail of how SPARK is applied varies from project to project, depending on factors which include the required integrity level, applicable standards, and the overall verification strategy for the system (in which SPARK will play a part amongst other techniques and tools). Together, these considerations will lead to a set of verification objectives for SPARK, which will be documented in the technical plan at the start of a project (and which in turn support the assurance case either implicitly or explicitly if it is a formal deliverable).

In spite of these variations, one can identify certain typical ways in which SPARK is applied on projects which have been shown to deliver high value in relation to the effort required. The table in Figure 1 summarizes Altran's experience of how best to apply the different assurance levels possible with SPARK vs. the relative design integrity level of the software under development. Stone level is not represented as it is more an intermediate level during adoption of SPARK than a target assurance level.

Software Integrity Level		SPARK Verification Objective			
DAL	SIL	Bronze	Silver	Gold	Platinum
A	4				
B	3				
C	2				
D	1				
E	0				

Fig. 1. Technical Planning Guidelines for the Application of SPARK. The filled cells denote the three most common categories of application.

The way to understand this table is as both a summary of experience on industrial projects at Altran UK and as a starting point for how Altran UK approaches new projects. Every project at Altran will tailor its own approach. However, one would expect new projects to fall within the typical region(s) indicated in the table; any which did not would require justification in the planning phase.

We have chosen to represent the software integrity level using two commonly understood scales: DAL (Design Assurance Level) is the terminology from DO-178 and SIL (Software Integrity Level) is the terminology used in DEFSTAN 00-55, IEC 61508, EN 50128 et al. The correspondence between DAL and SIL is necessarily informal because different standards define the levels according to different criteria. Note also that while DAL-E is defined by DO-178 its counterpart "SIL-0" is an informal but widely used term taken here to mean software below SIL-1 but which is still well-engineered.

Experience shows that projects can be grouped into three broad categories, shown by the three filled regions in the table. Category 1, shown in black, represents our practice at the highest levels of integrity SIL-3 and SIL-4. Within this category, Silver (AoRTE proof) is considered the "default level", but may be increased to Gold or even Platinum depending on whether key properties and functional correctness respectively are verified by other means. Targeting Platinum (full functional proof) becomes less likely for a SIL-3 system where verification by testing could more easily be argued to be sufficient.

Category 2, shown in gray, captures our practice at medium levels of integrity SIL-1 and SIL-2. Silver is still the default level, and it is very unlikely that Platinum would be employed on systems below SIL-3. However, proof of key properties (Gold) should still be strongly considered. There may be some key property where proof represents a very efficient means of verification, *i.e.* it is relatively easy to prove and relatively difficult to verify by any other means. The nature of such properties will vary from system to system, but could include even one key safety property ("the lift will not move when the doors are open") or security property ("the account details cannot be accessed when the user is not logged in"). While testing can provide some level of confidence in such properties it can never provide a complete guarantee for any realistically-sized system, due to the impossibility of covering all possible states and input combinations.

Category 3, shown in light gray, represents the lowest levels of integrity, so-called SIL-0. Even here, Silver is the default objective, but this could be weakened to Bronze if there is enough confidence that AoRTE was being sufficiently-well assured by other means or mitigation.

The table shows that - for all but SIL-0 software - SPARK code will as a minimum be checked for AoRTE. Note that this level of verification implicitly means that all SPARK code has also been shown to be free of references to uninitialized variables and basic data flow errors. Experience shows that the presence of this kind of flaw - which can have far-reaching consequences - can be immensely difficult to detect by other forms of verification [16].

A key part of the software engineering process which maximizes the benefit of SPARK is a careful delineation of the "SPARK boundary" *i.e.* choosing which parts of the application software will be written in SPARK. Although the benefits of SPARK would push towards maximizing the proportion of the software written in SPARK, other factors are likely to affect this engineering decision. For example, there may be pre-existing libraries to support the user interface or other external communication protocols that one wishes to use and which are qualified by alternative means. It is not unusual even to use different levels of SPARK verification within the same application. For example, SHOLIS [6] used this approach with SIL-4 parts of the application attaining full functional proof (Platinum level) while in lower-integrity functions (SIL-2) they verified only up to AoRTE (Silver level). The non-interference between different sections of the code was assured by the use of information flow analysis: a contract was attached to each subprogram specifying which global data items it could access in accordance with its SIL and the SPARK tools were used to check that the implementation respected these contracts. More generally, consideration has to be given to the assumptions that are made to support the verification objectives - how these are satisfied or mitigated by other activities in the overall V&V strategy [14].

The use of SPARK within the Correctness By Construction framework as described above has been demonstrated to produce software with very low defect density when compared to other high-integrity processes [6]. Although the above approach is the standard approach within Altran UK, the company has continued to explore new ways in which benefits can be gained from the use of SPARK, in particular the possibility of so-called "hybrid" approaches to verification, where a mixture of static and dynamic verification techniques are used to exploit the SPARK contracts.

The hybrid approach that Altran is currently pioneering, called ConTestor, uses SPARK verification at Silver level *i.e.* assurance of AoRTE using proof. In addition, SPARK contracts are used to add a functional specification to the code. Rather than verifying these contracts by proof using the SPARK tools (as per the standard Platinum approach), they are verified dynamically by testing. To perform these tests a fully-integrated version of the code is compiled with the run-time checks enabled for the functional contracts. Test cases for the integrated code are generated using constrained-random test generation and if no exceptions

are raised during execution then the code has passed this functional test. The contracts effectively provide a test oracle *i.e.* an independent calculation of the expected outcome for each test case. However, rather than having to manually calculate the expected outcomes per test case, the contracts are written once and provide an implicit definition of the expected outcome for all possible test cases.

4 The Adoption of Formal Verification

Contary to Altran UK, Thales has no established use of formal verification, but different units in Thales have been experimenting since 2015 with formal verification of programs using SPARK and Frama-C. The two case studies in this section describe the experiments with SPARK in the context of two different units working respectively in the domains of air defense systems software and cryptography.

4.1 First Study: Define an Adoption Strategy

One trait of established industrial software development processes is their inertia in accepting new practices which could be considered as too disrupting, either by lack of understanding and know-how or, mostly for early adopters, because of the difficulty to assess costs and benefits. In the latter case, the upfront adoption effort is hiding the longer term process optimisation opportunities. In order to get a first idea of the possibilities SPARK-based formal verification could provide at Thales, a study was carried throughout 2016 with the aim of producing a first set of deployment guidelines supported by real life experiments on actual software applications.

As formal verification with SPARK is not a widespread technology in the software industry, a prerequisite is to picture the range of its capabilities with a simple to remember concept. This led to the definition of the five levels of assurance previously introduced. As part of this study, AdaCore and Thales wrote a guidance document [1] describing how SPARK could be adopted at these different levels. This is further conditioned by the phase of the software development lifecycle, which has a significant impact on the definition of activities to be performed when deploying SPARK.

Given the current state of progress of some ongoing software development projects, four case studies were identified as potential targets for SPARK deployment experiments, from teams working on air defense systems software.

The first case study meant to assess the effort for transitioning from Ada to SPARK code (Stone level) using a mature software application about to be ported onto a new execution platform. As porting the application on a new platform using a different compiler may introduce a different behavior in case of errors such as references to uninitialized variables, reaching the Bronze level seemed a desirable aim. A significant refactoring effort was required in order to

cope with constructs excluded from the SPARK subset of the Ada language, the most prominent one being pointers. Thales engineers started using successfully the refactoring solutions described in the guidance document, but did not manage to complete refactoring in the expected time frame (5 person-days), due to the size of the chosen code base (around 300 klocs). This is expected to be completed in the coming year.

The second case study focused on programmer proficiency. In that case study the small subprogram of less than 10 lines of code listed in Figure 2 was given to an experienced Ada programmer with the goal of performing validation activities, both using the usual unitary test approach and a contract-based approach. Based on current tools, it took less than one hour for the experienced software test engineer to set up a working test environment for the subprogram. On the other hand, writing relevant contracts on that same subprogram to formally prove properties took an order of magnitude more time for the same engineer. Interestingly, the amount of code to implement a contract was in that case as long and complex as the code to prove. As a consequence, there is no intention to invest in Gold level verification on numerical computations in the near future. The lesson is that one should start with the lowest levels of assurance and work upwards, as practiced in the subsequent case studies.

```
subtype Nb_Type          is Natural      range 0    .. 100;
subtype D_Time_Type      is Float        range 0.0  .. 1_000.0;
subtype Delta_Time_Type  is D_Time_Type  range 0.0  .. 1.0;

procedure Study_Case (Nb_Of_Fp     : in     Nb_Type;
                      Nb_Of_Pp     : in     Nb_Type;
                      Delta_Time   : in     Delta_Time_Type;
                      Time         : in out Float)
with
  Pre     => Nb_Of_Pp > 0 and Delta_Time > 0.0 and
             Time ≥ 0.5 * Float (Nb_Of_Fp + Nb_Of_Pp) * Delta_Time and
             Time < Float'Last - Float (Nb_Of_Fp + Nb_Of_Pp) * Delta_Time,
  Post    => (if Nb_Of_Fp > 0 then Time ≥ Time'Old)
is
   D    : D_Time_Type;
   T_Fp : Float;
   T_Pp : Float;
begin
   D    := Float (Nb_Of_Fp + Nb_Of_Pp) * Delta_Time;
   T_Fp := Time - (D / 2.0);
   T_Pp := T_Fp + Float (Nb_Of_Fp) * Delta_Time;
   Time := T_Pp + 0.5 * Float (Nb_Of_Fp) * Delta_Time;
end Study_Case;
```

Fig. 2. Simple problematic case for formal verification. The initial postcondition contained a strict inequality `Time > Time'Old` that is not true for high values of `Time` where the offset is absorbed. Even the fixed postcondition with a non-strict inequality is not provable as it depends on the respective magnitures of `Nb_Of_Fp` and `Nb_Of_Pp` which are not specified in precondition.

The third case study was designed to complement test result artifacts on automatically generated code. A large amount of the unit's software application source code relating to data binary serialization and deserialization is automatically generated. The code generator compiles data models described through a domain specific language into Ada code. Up to now the test strategy for the code generator was mostly based on a limited set of regression tests and the confidence acquired over time as this technology was deployed across many projects over the last fifteen years. However, a hard to trigger weakness was lying dormant, which was cleaned up using a Gold level approach. With the support of SPARK experts, a first stage was to correct and refactor the code (2 kloc for the runtime and 21 kloc of generated code) to pass Stone, Bronze and Silver levels. For code written by savvy programmers making a moderate use of specialized language features such levels are easy targets, in this case less than half a person-day for a few hundreds lines of code. Reaching Gold level to prove one property related to buffer overflow on the generated code required a larger effort, two person-days in that case, in order to refactor the code for proof (to avoid the weakness mentioned above related to buffer overflow), interact with automatic provers through intermediate assertions and provide the required loop invariants. Given the extra level of confidence regarding the robustness these changes provide, Thales plans to deploy them in the next release of the code generator.

The fourth case study targeted the proof of safety properties in a context where safety standards apply. Safety properties are usually written as "nothing bad will ever happen" and, since their scope is usually on a large part of the code, need to be specified at the highest level of the code, almost at the entry point. Inside a 70 kloc control commands project, Thales and AdaCore engineers identified a few units (7 kloc) defining a set of high level automata where those properties could be specified. As a first step, the engineers reached the Stone, Bronze and Silver levels on this code in less than a person-day. Then, contracts were added on subprograms implementing the automata, mostly to express the effect of calling each automaton, also in less than a person-day. Automatic proof was obtained without much difficulty after that, with no need for intermediate assertions, loop invariants or specific proof switches. The lessons learned here are that SPARK is expressive enough for typical safety automata properties, and powerful enough for automatic proof of such properties.

Lessons learned. From an adoption point of view, Thales concluded from this first study that formal verification as implemented by SPARK 2014 and its associated toolset can be considered as a toolbox providing various opportunities for subsetting and tailoring. This flexibility gives the possibility to fine-tune the gradual insertion of formal verification techniques in existing processes, while mitigating risks both on their efficiency from a cost and planning point of view and their ability to output software with a defect density under control.

4.2 Second Study: Implement and Refine the Adoption Strategy

In the field of high-security applications, which is particularly important for Thales, testing represents a considerable part of the software development process. In addition to unit tests, other principles are implemented such as enforcing coding rules, peer code reviews and qualimetry surveys with many tools checking that those principles are strictly followed. One solution to lighten and improve this process, to produce software of improved quality, is the use of more suitable tools, such as formal verification tools to replace part of the tests. Indeed, formal proofs allow a comprehensive checking of proved parts, unlike testing that can only guarantee a partial checking of the software.

After a previous internship in 2015 comparing some available environments for formal verification (eCv [9], Frama-C [18], SPARK), another six-months internship in 2017 was dedicated to the study of the benefits of the Ada language and particularly the SPARK language for the security software developed at Thales. During this internship, Thales evaluated the various advantages of Ada and SPARK, by implementing the Adacore and Thales adoption guidance on two proofs of concept in the field of cryptography.

The first case study was porting from C to Ada, then to SPARK, part of a cryptographic library which is used as an abstraction layer between a lower level cryptographic library (also in C) and client applications. This case study followed the guidance document produced in the previously mentioned first study, to convert an application from Ada to SPARK.

The preliminary stage consisted of porting the C library code to valid Ada code. Porting API (.h files) was facilitated by g++ switch "–fdump-ada-spec" which produced comprehensive Ada specifications (.ads files) as well as Ada body skeletons (.adb files) generated automatically with the gnatstub tool. The body code was completed manually without difficulties as most C idioms are available with Ada. Interfacing with the C low-level cryptographic library was essential and was supported natively by Ada. This small step brought simpler code with pointer-related defensive code eliminated thanks to the use of handy Ada array attributes and warnings from the Ada compiler.

Firstly, Stone level was reached by transforming the Ada code to be valid SPARK code. It mostly consisted in suppression of pointers (or at least encapsulating them in a non-SPARK unit) and transformation of functions with side effects into procedures (or at least encapsulating them in wrappers within a non-SPARK unit). Thus, it was possible to make a first analysis of the code with SPARK tools. This first step didn't require major changes in the code but it pinpointed parts of the code with potential security vulnerabilities (pointer casts and side effects in particular).

In a second stage, Bronze level was reached, analyzing the code for data flow and variable initialization. Data flow (Global) and information flow (Depends) contracts were added in the code to specify precisely the intended behavior. The analysis detected unused inputs which could then be removed, which is useful

for maintenance, as well as partially initialized data structures, which is useful for debugging.

In a third stage, Silver level was reached, ensuring absence of run-time errors in the code (AoRTE). Preconditions were added in the code, mostly to link the right algorithm with the right variant of a discriminated structure.

In a fourth stage, Gold level was reached, verifying the functional behavior of the code. Preconditions and postconditions were added in the code to specify key security requirements: cleanup of security-sensitive working variables, correctness of output value, and consistency between parameters as presented in Figure **??**. At this level, all the existing defensive code had been replaced by contracts. By achieving complete proof of these specifications, the propagation of error codes from low-level subprograms to high-level ones was no longer necessary.

```
procedure computeSha (input          : in      uint8_t_array;
                      inputByteLen   : in      stdint_h.uint32_t;
                      digest         :    out uint8_t_array;
                      outputByteLen  : in      stdint_h.uint32_t;
                      hashByteLen    : in      stdint_h.uint32_t)
with
  Contract_Cases => (hashByteLen = 20 => digest'Length = 20,
                     hashByteLen = 32 => digest'Length = 32,
                     hashByteLen = 48 => digest'Length = 48,
                     hashByteLen = 64 => digest'Length = 64),
  Pre => (inputByteLen = input'Length and
          (hashByteLen = 20 or hashByteLen = 32 or
           hashByteLen = 48 or hashByteLen = 64));
```

Fig. 3. Simple case of contract for expressing consistency between parameters, checking here that the length of the hashed message `digest'Length` is consistent with the type of hash used `hashByteLen`.

The second case study was about producing an API similar to the API ported from C during the first proof of concept, this time based on a low-level cryptographic library in Ada, which was also later proved with SPARK. The whole process from Stone level to Gold level was followed again. New technical issues arised: the need for loop invariants, contracts on type hierarchies for subprograms supporting dispatching, visibility of global variables in contracts of high level subprograms, and non-provable Ada code. Though loop invariants are the basis of formal proof, they are considered as tricky. Many unproved properties came mostly from weak preconditions or weak postconditions of subprograms called inside a loop, which were not obvious to understand. Object Oriented Programming brings another layer of complexity, with specific rules for inheriting subprograms and contracts over these subprograms. Global variables mentioned in data flow contracts propagate to the upper levels of the call tree, where they may not be visible anymore (due to abstraction mechanisms in Ada), which required costly workarounds. A better solution would have been to hide this particular effect in a low-level non-SPARK package body, or to use the data

abstraction feature available in SPARK. Finally, some idiomatic Ada code did not lead to automatic proofs in SPARK, which led to changes for simpler and more readable code.

Lessons learned. Thales learned a few lessons from this second study. First, the adoption guidance document was really helpful: it eased the implementation of SPARK during the second internship. As a result, it was also refined for future uses inside and outside Thales. Secondly, as stated in the guidance document, *Users should refrain from changing the program for the benefit of only getting fewer messages from the tool*, a principle that could be phrased as "do not please the tools". Of course, it is sometimes adequate to change the program in a way that will cause some messages about unproved properties to disappear, provided this favors code quality, readability or maintenance. Otherwise, tools provide ways to silence messages, that should be used instead of changing the program. Thirdly, reaching Gold Level is more easily achievable when clear and meaningful software specifications are available. Finally, not all code can be proved but non-provable parts that are well identified can undergo peer code reviews. For instance, 90% of the code was automatically provable in the second case study, after suitable addition of contracts where necessary.

5 Related Works

Formal methods have long been considered as a means of compliance to satisfy verification objectives in critical software development for some certification domains, for example in railway (EN 50128) and industrial processes (IEC 61508). The avionics standard DO-178C in 2012 has more recently recognized formal methods as a means of compliance on a par with the dominant technique of testing. Other certification standards in the domains of automotive (ISO 26262), nuclear (IEC 60880) and space (ECSS-QST-80C) also recognize some uses of formal methods as verification techniques.

The adequateness of formal methods for certification was thoroughly investigated by John Rushby in his report for the NASA in 1993 on "Formal Methods and the Certification of Critical Systems" [26]. Although Rushby's report talks about "formal methods", this mostly corresponds to what we call today "theorem proving": model checking techniques are mentioned en passant, and nothing is said about abstract interpretation techniques, which did not have then the recognition that they do today. More recently, Graf and Garavel studied extensively the use of formal methods for developing critical systems, and they cover in particular the impact of formal methods on development and verification processes [13]. More specific guidance exists in certain application domains, such as in avionics [3].

There is on the contrary very little guidance on the use of specific formal methods and tools. This is somewhat remediated by the availability of tool specific user guides and publicly available experience reports [28]. Company-specific guidance is developed to carry-over the experience gained from project to project,

in the companies using formal methods, but such guidance is kept confidential. Indeed, the experience gathered through previous projects is considered as a business advantage over the competition, and the guidance having been developed in the specific business context of the company, the information related to formal methods usage is very tied to other confidential information. In their dual role of SPARK tools providers and practitioners, Praxis and then Altran have always been keen on publicizing best practices and lessons learned with formal verification on industrial projects [17,7,8]. The publication of the guidance co-developed between AdaCore and Thales [1] on SPARK adoption follows this lead, which was possible because it was written since the start with the tool provider. This is similar to the joint effort by tool provider CEA, certifier Bureau Veritas and user Sirehna to publish guidelines on the use of Frama-C [11].

Formal methods have been divided between heavyweight and lightweight ones, with the former being the original formal methods and the latter also being called the *disappearing* formal methods [27]. SPARK is a case of lightweight or disappearing formal methods, in which the user does not directly manipulate the underlying formalism, but instead interacts with tools through multiple interfaces. Formal methods and tools are usually placed at some point in between the heavyweight and lightweight extreme points. With the notion of software assurance levels, we have shown that a given tool can be placed at multiple places along this axis, and that a project can move between these places using the same tool.

In particular, it is likely that other formal verification platforms for software such as Atelier B and Frama-C could similarly define their own software assurance levels. For example, the plugin structure of Frama-C could be used to define levels in terms of plugin usage [18].

6 Conclusion

Formal program verification with SPARK has been used for years at companies like Altran UK to get as close as possible to zero-defect software. Altran UK has developed software engineering processes to maximize the costs-benefit ratio of using SPARK. In particular, it has defined a mapping between levels of use of SPARK and software assurance targets (SIL/DAL), which is used by all projects at Altran UK. Altran UK is now investing in its use of SPARK for the future, by investigating innovative ways to generate tests from contracts, to combine tests and proofs and to analyze code generated from Simulink.

Other companies like Thales are starting to use SPARK to obtain similar benefits. We have presented in this article the lessons learned at Thales on various deployment experiments at different levels of use of SPARK. As for every promising but complex technology, the success of its deployment is conditioned by the pace at which adopters can climb the learning curve and identify relevant insertion points and strategies into established development processes. While AdaCore expertise was essential in the success of these experiments, Thales has identified typical use cases where the methodology used could be replicated with-

out external help. Thales is now aiming at clarifying how SPARK can be adapted to its internal processes. The guidance document written as a result of Thales experiments is being used to support adoption of SPARK in other teams inside Thales and is available for other companies to start on this path.

We have benefited in multiple ways from the definition of the five software assurance levels that can be achieved with SPARK. First, the five levels clarify the verification objectives that can be achieved with formal verification: not only they provide simple and easy-to-remember names for communicating between stakeholders, they also make it explicit that upper levels build on the lower levels, and they provide at each level a clear identification of the costs and benefits. Secondly, the five levels make it easier to plan for progressive adoption of higher levels of software assurance, with lower levels requiring less effort than higher levels, and each level providing already very valuable benefits. These results could be translated to other formal methods that similarly provide different depths of use that could be translated to assurance levels.

Acknowledgements. We would like to thank the anonymous referees for their useful remarks, as well as our colleagues at AdaCore, Altran and Thales for their reviews on earlier drafts of this article.

References

1. AdaCore and Thales. Implementation guidance for the adoption of SPARK. https://www.adacore.com/books/implementation-guidance-spark.
2. P. Baudin, P. Cuoq, 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.
3. Duncan Brown, Hervé Delseny, Kelly Hayhurst, and Virginie Wiels. Guidance for using formal methods in a certification context. In *Proc. ERTS*, 2010.
4. Patrice Chalin. Engineering a sound assertion semantics for the verifying compiler. *IEEE Trans. Software Eng.*, 36(2):275–287, 2010.
5. Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 342–363, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
6. Roderick Chapman. Correctness by construction: A manifesto for high integrity software. In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, SCS '05, pages 43–46, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
7. Roderick Chapman. Correctness by construction: A manifesto for high integrity software. In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, SCS '05, pages 43–46, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
8. Roderick Chapman and Florian Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 17–26, Cham, 2014. Springer International Publishing.

9. David Crocker. Can C++ be made as safe as SPARK? In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 5–12, New York, NY, USA, 2014. ACM.
10. Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. Rail, space, security: Three case studies for SPARK 2014. In *Proc. ERTS*, 2014.
11. Lucas Duboc, Sébastien Flanc, Florent Kirchner, Hélène Marteau, Virgile Prévosto, Franck Sadmi, and Franck Védrine. Safer marine and offshore software with formal-verification-based guidelines. In *Proc. ERTS*, 2016.
12. Manuel Fähndrich. Static verification for code contracts. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, pages 2–5, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
13. Hubert Garavel and Susanne Graf. *Formal Methods for Safe and Secure Computers Systems - BSI Study 875.* BSI German Federal Office for Information Security, 2013.
14. Johannes Kanig, Rod Chapman, Cyrille Comar, Jerome Guitton, Yannick Moy, and Emyr Rees. Explicit assumptions -a prenup for marrying static and dynamic program verification. 07 2014.
15. James Cornelius King. *A Program Verifier.* PhD thesis, Pittsburgh, PA, USA, 1970. AAI7018026.
16. S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof more cost-effective than testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, Aug 2000.
17. Steve King, Jonathan Hammond, Roderick Chapman, and Andy Pryor. Is proof more cost-effective than testing? *IEEE Trans. Software Eng.*, 26(8):675–686, 2000.
18. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.
19. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.
20. Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014. In *7th International Symposium on Leveraging Applications*, 7th International Symposium on Leveraging Applications, page 16, Corfu, Greece, October 2016. Springer.
21. K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*, 2010.
22. Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
23. John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK.* Cambridge University Press, 2015.
24. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.
25. J. Rushby. Software verification and system assurance. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 3–10, Nov 2009.
26. John Rushby. Formal methods and the certification of critical systems. Technical report, 1993.

27. John Rushby. Disappearing formal methods. In *High-Assurance Systems Engineering Symposium*, pages 95–96, Albuquerque, NM, nov 2000. Association for Computing Machinery.
28. 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.