Safe and Secure Autopilot Software for Drones

Amin Elmrabti, Valentin Brossard, Yannick Moy, Denis Gautherot, Frédéric Pothon

Safe and Secure Autopilot Software for Drones

Amin Elmrabti , Valentin Brossard[2] , Yannick Moy[3] , Denis Gautherot[1], Frédéric Pothon[4]

Sogilis, 4 Avenue Doyen Louis Weil, 38000 Grenoble, France.

[2]**Hionos**, 12 Avenue Des Près, 78180 Montigny Le Bretonneux, France.

[3]**AdaCore**, 46 Rue d'Amsterdam, 75009 Paris France.

[4]**ACG Solutions**, 19 Chemin du triol, 34380 Viols Le Fort, France.

***Abstract*-- We are interested in the problem of providing safe and secure software for drones to make them as safe as airplanes. As there is no certification standard for most commercial drones that exist today, we have chosen the well-known DO-178C/ED-12C avionics standard as a framework to define a suitable formal development and verification process. Our process is based on this standard and its three supplements: DO-331/ED-218 (model based development), DO-332/ED-217 (object oriented technology) and DO-333/ED-216 (formal methods). Our process is an original adaptation of the usual V-cycle based avionics processes to an iterative and incremental environment, where development and verification are performed in short increments. In this paper, we describe how we achieve that in practice and discuss in more details the integration of the formal methods supplement.**

I. Introduction

The rapid increase in the number of civilian drones, for both leisure and commercial use, poses significant threats to the safety of the general public. The authorities' solution so far has been to impose severe restrictions on the use of drones. International committees have been formed to discuss the normalization of the drones operations (JARUS, Eurocae WG 105, GUTMA, EASA, *Conseil Pour Le Drone Civil* in France). Current discussions lean towards requesting a safety assessment for the *Specific* and the *Certified* categories, thus demanding that a few development and verification activities are enforced. Today, drones are classified based on operations risks concerns (figure 1) [1]. In the open category, where the level of risk is low, the required level of safety will be ensured through a set of requirements and functionalities. In the Specific category, the safety will be ensured through a standard risk assessment process. In the Certified category, the risk is similar to current manned aviation operations, and safety is ensured with traditional safety measures and processes (certification and licensing).

Figure 1. Drone classification

Autopilots that come from the world of model making have been developed without a specific focus on safety, which leads to bugs and unexpected behaviors in the software. Because of their intended use, commercial drones need safe and flexible autopilots; we should be able to prove a high level of safety and easily adapt it to different kinds of operations. The application of DO-178C standard suite for avionics certification in this domain is a promising option to develop software for drones used in certain types of applications, especially in urban areas. But the application of DO-178C is traditionally very costly. Thus, we should provide an innovative approaches to DO-178C compliance to avoid unacceptable costs impacts while increasing the reliability of the drone software.

We aim to achieve much higher levels of assurance in our next generation of autopilots called Pulsar Flight System (PFS) [2] by using an efficient development process and formal methods when possible. Our decision to rely on

formal methods is partly related to the adoption of the formal methods supplement in the latest version of the avionics certification standard. This merely acknowledges the fact that formal method technologies do exist which can ease the development of avionic software. More important for us is that some formal methods and technologies like SPARK [3] have shown to lead to high assurance software.

In this paper, we will introduce the software lifecycle and some of the methods selected for drone autopilot software to achieve the applicable DO-178C/ED12C objectives at the most critical level (A). We will describe how the software certification standard used in avionics can be used in the drone software industry to achieve a high level of assurance through an innovative use of the standard and its supplements. We will discuss particularly the ability to use formal methods on software.

The rest of the paper is organized as follows. Section II reviews some related works in relation to the development of safe software for drones. In this section, we will present also the technologies used in formal methods such as SPARK. Section III introduces our proposed development and verification process and tools. In this section, we detail how we've put into practice the three supplements: DO-331/ED-218 (model based development), DO-332/ED-217 (object oriented technology) and DO-333/ED-216 (formal methods). Section IV presents the progress of our work and discusses the advantages and the disadvantages of this process, particularly in relation with the Formal method tools. Section V concludes this paper and presents outlooks on future works.

II. Related Works

The Formal method techniques have been used in the development of critical embedded software, in avionics [4], railways [5], medical devices [6], etc. In [4], the authors describe how they used formal method tools for unit proofs, WCET (Worst Case Execution Time) analysis, and for maximum stack usage computation of C code to cover some DO-178B objectives. Frama-C [7] is a toolbox framework that aims to analyze C programs. It is extensible with plug-ins which implement a specific analysis. Jessie [8] is an example of a plugin for the Frama-C environment, aimed at performing deductive verification of C programs.

SPARK [3] is a subset of Ada. It adds contracts and type invariants as part of regular Ada code. SPARK is dedicated to real-time embedded software that requires a high level of safety, security, and reliability. SPARK 2014 is the last major version of SPARK, designed to interpret Ada 2012 contracts. SPARK has been used for decades to achieve high assurance software in domains subject to certification, such as military and avionics. As a preparatory step for the CAP2018 project [9], AdaCore used SPARK to rewrite the code of a small open source leisure drone called Crazyflie [10], originally written in C, to demonstrate absence of errors in drone software [11].

In [12] [13], a comparison is made between the formal method techniques, particularly Frama-C and SPARK. The two technologies are quite similar in their code functionality. Both of these technologies use contracts and assertions. In this project, we chose to use SPARK because the SPARK syntax is part of the Ada 2012 language. There is no need to integrate a new notation. The Ada language allows us to create subtypes and SPARK's static analysis will verify the subtypes outside range and prove the overflow in the subtype's values.

More recently, SPARK has been used by small teams of a few developers to develop high assurance software of satellites [14] [15] and gliders [16], with an ad-hoc development and verification process more akin to agile than to DO-178. Other teams have adopted an agile approach to avionics software development and verification [17]. Ways to combine agile methods and formal methods have also been explored [18]. The CEOS project [19] aims at developing a reliable and secure system of inspections for pieces of works using professional mini-drone. CEOS does not target certification, but rather building safety cases for the Specific category of drones as planned in the upcoming European regulation.

III. An Agile and Certifiable Development Process

We built the software life cycle around an iterative and incremental process. The incremental approach consists of developing and verifying progressively a limited number of features. This method allows us to provide a functional subset for system integration, and thus to get a rapid feedback on the realized features and early error detection and correction.

A. Agile Development and Verification Process

Our development process is organized around a set of tickets. Every artefact required by DO-178C corresponds to a ticket. Every code, test or requirement modification corresponds to a ticket. We identified 9 types of tickets, each of them representing a set of activities to perform.

(1) Type 1: Process or System Improvement: These tickets are used for every evolution of the certification plans or standards (coding standard, modelling standard…). For example, if we want to modify the way we are managing the configuration of the software that will end up in a modification of the SCMP (Software Configuration and Management Plan). This modification will be traced through a Type 1 ticket. Any changes of system requirements are also tracked in Type 1 ticket. Any change is analyzed and any necessary child tickets are opened to implement it. **(2) Type 2: Software Requirement:** Type 2 tickets are gathering all activities needed to create and review Software Requirements. The activities here correspond mostly to the writing of Software Requirements that comply with System Requirements and the review of these written requirements. At the same time, Software Tests (Test cases used to verify Software Requirements) are written. **(3) Type 3: Architecture:** An initial architecture is defined in the scope of a first Type 3 ticket, based on a first set of functions defined in software requirements. Next Type 3 tickets are raised, based on new or modified software requirements that will adapt and enrich the architecture. Type 3 ticket is used to define dynamic architecture (tasks, frequencies) and components. Components have inputs and outputs (not necessary black box input / output) and are linked together to perform a feature described by one or several Software Requirements. **(4) Type 4: Component Requirement:** Thanks to the architecture defined in Type 3, we can write requirements defining the behavior of components. These are Component Requirements. Component Tests are also written in this ticket. **(5) Type 5: Source Code:** Activities under this Type 5 ticket are Test Procedure and Source Code development. It includes the full verification of the components, including the development and execution of component tests procedures. **(6) Type 6: Integration:** Under this type of ticket, the Source Code is compiled and Software Tests are run on the target computer containing the whole software. **(7) Type 7: PDI Instance:** PDI stands for Parameter Data Item, it represents all the data that are used to configure the software. This ticket is used to develop and verify a new instance of a PDI. Structure and attributes of PDI are developed in the scope of Type 2 ticket. **(8) Type 8: Problem Report:** Every deviation from the process must be recorded in a Type 8 ticket before it can be handled. Each Type 8 ticket is analyzed, and the necessary changes are identified in one or several child ticket. **(9) Type 9: Delivery:** Before a delivery, a Type 9 ticket representing all activities to perform shall be created. In the scope of this ticket, in application of continuous integration, the complete set of test procedures are re-performed.

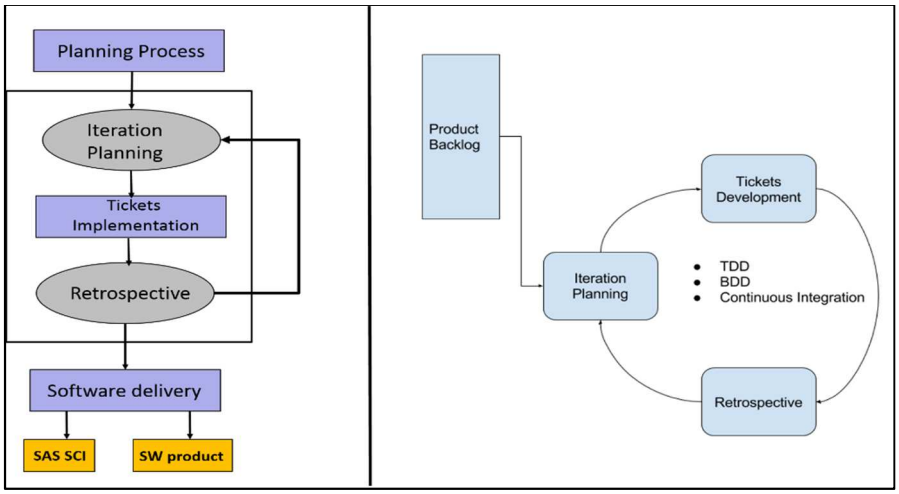

Figure 2. Agile Development Process

For a single feature at System Level, we will usually use one Type 2, one of Type 3, one of Type 6 and several of Types 4 and 5 tickets. When the Type 2 ticket is done, the Type 3 ticket can be created and so on and so forth: There are dependencies between tickets. Moreover, independence is required for several activities by DO-178C. For example, if a person performs all the activities of a Type 4 ticket, she shall not perform activities under the corresponding Type 5 ticket.

A classical iteration is composed of several tickets of different types, representing enough work for the whole team for the duration of an iteration. These tickets are chosen by the team according to priorities during iteration planning. At the end of the iteration, a consistent (and verified) set of features is added, and an iteration retrospective is conducted with quality assurance.

In addition to this iterative and incremental process, we also use a couple of agile best practices like TDD (Test Driven Development), BDD (Behavior Driven Development) and continuous deployment. TDD and BDD are interesting by the fact that tests (Component Test for TDD and Software Test for BDD) are written before the

code, so that the code architecture and development are not impacting test development. Tests are written in Type 2 and Type 4 tickets (respectively, Software Tests and Component Tests) whereas Source Code is written afterwards, in Type 5 tickets. Continuous deployment allows us to be sure that at every time, all tests that have been previously developed pass. Each time a modification is done, the software is built, loaded on the target computer, tests are performed and source code coverage is checked.

Our development process uses the 3 supplements of DO-178C. It takes place in Type 4 and 5 tickets that can be developed using three different methods. The choice of the method is made at the component level. A component is developed using a single development method.

Natural Language: In that case, Component Requirements are written with natural language and implemented in Ada 2012 using object oriented technologies. Vulnerabilities mentioned in the DO-332/ED-217 supplement are addressed.

Model Based: Tickets can also be implemented using Simulink Models instead of Component Requirements, then the Source Code will be generated automatically using the QGen code generator. QGen will be qualified as a TQL-1 tool according to DO-178C classification, which allows us to trust the code generation. The DO supplement to be applied in this case is the DO-331/ED-218.

Formal Methods: The last solution to develop Type 4 tickets is Formal Methods using the SPARK language and prover. Component Requirements are written using SPARK Contracts, then Source Code is developed and the prover verifies the consistency between the Contracts and the source code. The formal analysis is used to replace component tests. The corresponding supplement is DO-333/ED-216.

For the next two sections, we will focus on a specific part of the process. We will present the main activities for component specification and implementation (Tickets 4 and 5).

B. Natural Language Requirements Process

The component requirements (equivalent to LLR a.k.a. (Low Level Requirements)) are written as text in natural language with a tool called RMTool. A textual requirement template is defined in the requirement standards and describes the structure of a requirement with a title, a set of inputs, outputs and a body which details the behavior of the requirement and how we compute the inputs to generate the outputs.

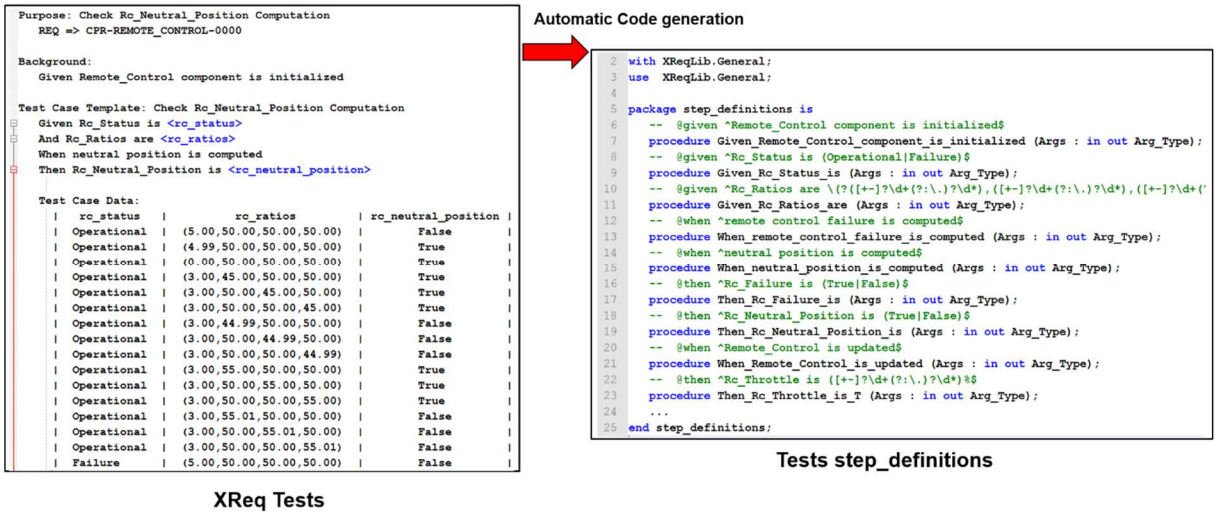

Figure 3. Test cases and step definitions code generation with XReq

RMTool maintains the traceability links and trace data between LLR and HLR (High Level Requirements) on one side and between the LLR and the associated tests, on the other side. Once the requirement is written, the tests are then described, before the code, since the TDD approach is applied in our development process. For these components the code is manually implemented in Ada 2012/ /or its subset SPARK 2014. The tests are written with the XReq test framework, which allows us to write tests using a language close to natural language. XReq also allows us to automatically generate test code (figure 3), particularly tests skeletons and steps. The development of the component requirement in a natural and textual mode is realized in a ticket of Type 4. The implementation is performed manually and the code is written in Ada 2012 in a ticket of Type 5. The test steps previously generated

with XReq are completed and executed. The compliance of the source code to the coding standards is checked. This verification is automatically performed with GNATcheck and then completed with a custom checker for the rules that are not verified by the tool.

C. Formal Methods Techniques and Development Process

The SPARK development and verification environment from AdaCore has been chosen for creating the code of the autopilot, as well as for expressing some of the LLR. This choice is based on its successful history in high-integrity domains, combined with the recent evolutions to democratize the use of formal methods with SPARK 2014. A major roadblock preventing the adoption of formal program verification for avionics certification is the absence of a general consensus on how to apply DO-333, despite significant dissemination efforts from the committee that developed it. We have defined a detailed process for the use of SPARK to satisfy the objectives of DO-333 as a replacement for certain forms of testing and reviews, with a focus on checking that the source code is consistent, accurate and complies with low-level requirements.

SPARK code is unambiguous, so the consistency of the source code can be analyzed automatically to show that it is free from reads of uninitialized data, arithmetic overflows, other runtime errors and unused computations (variables, statements, etc.). This covers objective 6 of table FM.A-5 from DO-333. We are completing that analysis with focused reviews. One of the main benefits of SPARK is that it automatically shows the source code complies with LLR expressed as function contracts. This covers objective 1 of table FM.A-5 from DO-333. For the modules which we will choose to express LLR as contracts, we will skip testing if the code implements the LLR. Function contracts can also express data dependencies and the SPARK toolset can automatically show that source code complies with this part of the software architecture. SPARK code is verifiable and conforms to a (programming language) standard by design. The source code of a function is implicitly traced to the LLR expressed in the function contract. Altogether, these cover objectives 1 to 6 of table FM.A-5 from DO-333.

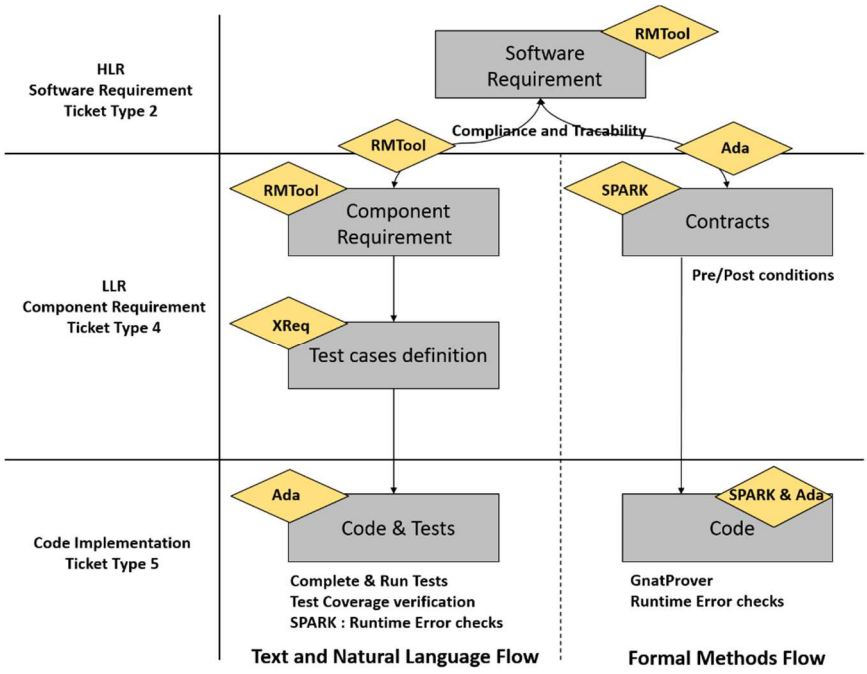

Figure 4. Code Development Process

The objectives of compliance and robustness of EOC (Executable Object Code) with respect to LLR (objectives 3 and 4 of table FM.A-6 from DO-333) can be addressed by relying on the source's code's corresponding objectives, assuming that one also provides a demonstration of property preservation between source code and EOC. One way to show property preservation would be to demonstrate with reasonable confidence that in all possible cases the compiler preserves the semantics of programs from source code to EOC. Unfortunately, no reasonable approach seems to be able to provide that confidence. Instead, we run integration tests in a mode where contracts are executed in SPARK, to gain confidence in the compiler properly preserving the semantics of source code in EOC; if it doesn't, the contracts proved in individual functions will (with very high probability) fail during integration tests. By running integration tests both with and without contracts being executed, and checking that

the outputs are identical, we can be confident that compilation of contracts does not impact compilation of code because otherwise, the outputs would most probably be different in some tests.

In our process, in the tickets of Type 4, the component requirements are written as a SPARK contract specification and no tests are written. The contracts are written in the specification files ".ads" (it is the equivalent of the header files ".h" in C language) which are part of the source code. The tractability between the software requirements and the corresponding component requirements, which are written as SPARK contracts, is described in the source code (the .ads), with the contracts as a set of comments.

In ticket of Type 5, the code is written manually and the GNATprove tool is used to detect all cases where the contracts are not satisfied. The analysis includes the verification that subprogram overriding respects the Liskov Substitution Principle (contract of the overriding subprogram describes sub-behaviors of the contract of the overridden subprogram). It also includes checking for the absence of runtime errors. SPARK could also be used for checking absence of runtime errors in the development flow based on natural language requirements.

IV. Implementation and Results

In this section, we present our advancement in the use of the formal process that we have defined, including formal methods with SPARK, in the development of the Pulsar Flight System, which is a reliable autopilot system for drones, compliant with the DO-178C aeronautical standard. We intend to use it for drones in places where safety is critical. We present how we use formal methods in our process with a case study about the "Battery_Manager" component of Pulsar Fight System for which the architecture has been defined during the Ticket Type 3 activities, as shown in the figure 5. Functionally this component must be able to recover from an abstraction layer "Battery_Interface" the current level of the battery as well as its operational status, in order to make this information available to the "State_Machine" component for further processing.

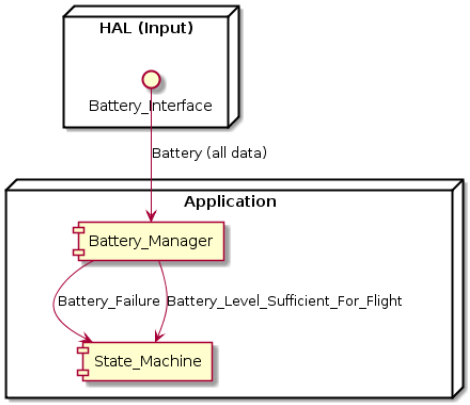

Figure 5. Ticket Type 3: Software architecture definition

A. Component Requirement: Type 4 Ticket Activities

As described in section III, the main goal of Type 4 ticket is to write the component requirements and tests descriptions. To put into practice formal methods techniques, we used SPARK contracts (otherwise known as "SPARK Functional Specifications") to describe the component requirements. This activity consists of initializing the component's package source code and writing all SPARK 2014 / Ada 2012 code and contracts. Contracts are written using pre/post conditions notation:

(1) The pre-condition is a predicate which expresses a constraint on the imported variables of the subprogram (e.g. that a scalar import is in a certain range, that an array is ordered, or perhaps some more complex properties).

(2) The post-condition is a predicate which expresses a relation between the initial values of imported variables and the final values of exported variables of the subprogram (when a variable is both imported and exported, we use some notation to distinguish its initial and final state in the post-condition.)

Currently, the SPARK subset does not allow pointers. In the coding standards that we defined, we decided to instantiate all the software components from the start of the application. Then we access all these components through pointers ("Access Type" in Ada). This allows the allocation of the memory once at the startup of the application, then no memory will be allocated or deallocated. This technical choice allows us to address the

dynamic memory management vulnerability mentioned in the DO-332/ED-217 supplement (Object Oriented Techniques Supplement). Considering this, we had to pay attention to the internal component architecture and how it will be integrated in the overall Pulsar Flight System application.

In this work, and in order to guarantee compliance with SPARK implementation, we have set aside "object-oriented" concepts for this component and went to a classic "Abstract Data Types" implementation. The internal data of the components was handled as SPARK “Abstract_State”. The state abstraction of a package specifies a mapping between abstract names and concrete global variables defined in the package. State abstraction allows the definition of Subprogram Contracts at an abstract level that does not depend on a particular choice of implementation, which is better both for maintenance (no need to change contracts) and scalability of analysis.

Figure 6. Ticket Type 4: Abstract State & Ghost Code

We noticed that it’s difficult to write clean and concise SPARK contracts without the need to write some “Ghost code” in order to make SPARK contracts more readable. Ghost code is additional code useful for the specification and verification of intended properties of a program, identified with the aspect Ghost, discarded during compilation, and has no effect on the functional behavior of the program. The Ghost method “Is_Battery_Level_Low_More_Than_Confirm_Duration” is declared in the specification file (see .ads in figure 6, line 12) and implemented in the package body file (see .adb in the figure 6, line 27). This Ghost method is called on the contract definition.

After having written the SPARK contracts (see figure 7 from line 18 to line 35, which correspond to two component requirements), we realized that the person writing the specifications is forced to write code (.ads for specifications and .adb for body code). To write code, the same person needs to take a look at the architecture defined in Ticket 3. Thereby, persons responsible for writing specifications need to have multiple skills (functional, coding, technical architectural ...). Figure 7 shows SPARK contracts specification code.Traceability toward the HLR (Software Requirement) is done with Ada comments. In figure 7, an example is illustrated:

- At line 23 we created a post-condition with a first conjunct which refers to *SWR-FAULT-0000* and *SWR-FAULT-0001* software requirements.
- At line 29 we added to the post-condition a new conjunct which refers to the *SWR-MANUAL-0000* software requirement.

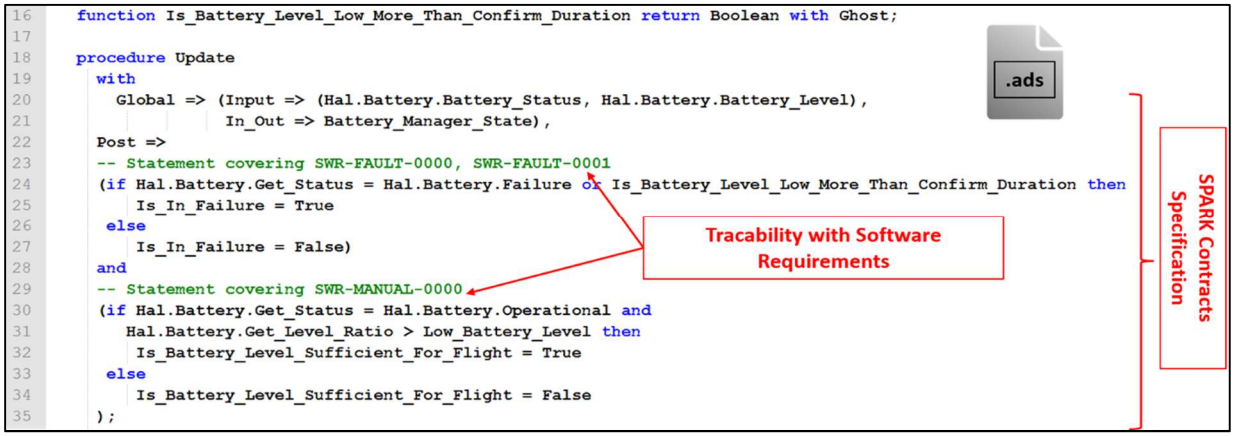

Figure 7. Ticket Type 4: SPARK Contracts specification

These two component requirements have been placed into one post-condition with the use of “if...then...else” expressions. Finally, the definition of the constants “Low_Battery_Level” and “Low_Battery_Confirm_Duration”, which could only be put in the component’s body package (as they are internal to this component), had to be moved into the component’s specification package and made public so that they could be used in SPARK contracts. The review activity was done based on this Ada package specification and the corresponding Type 2 ticket requirement.

B. Component Implementation: Type 5 Ticket Activities

In this activity, we completed the implementation of the component’s package body, as illustrated in figure 8.

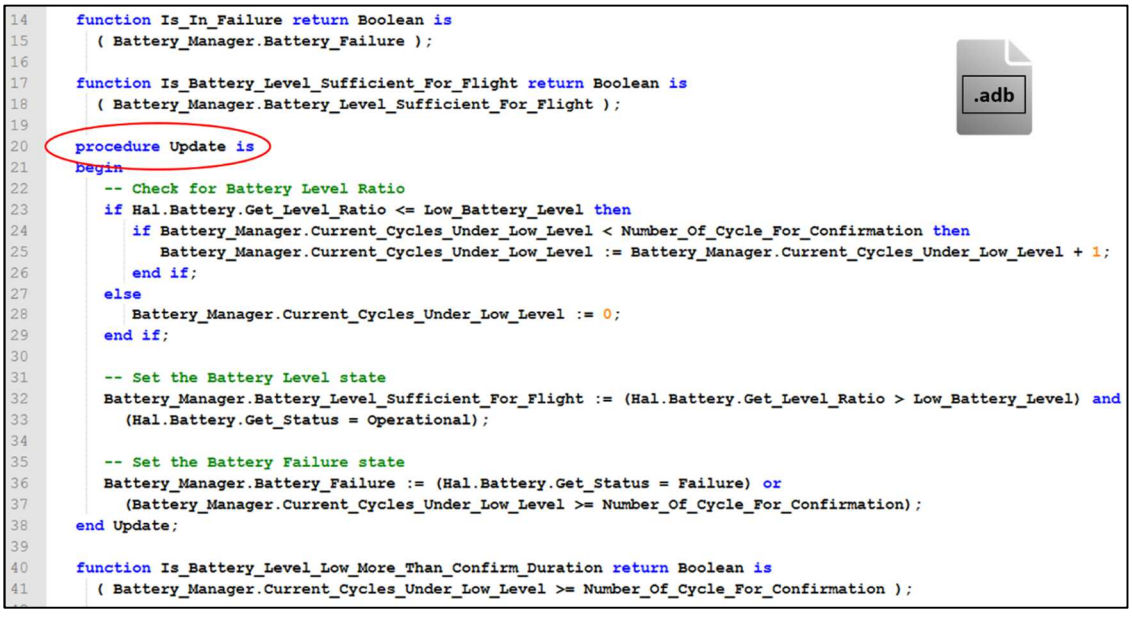

Figure 8. Ticket Type 5: Component implementation

Once the implementation is complete, code analysis can be performed using the GNATprove too to demonstrate the compliance of the implemented code with the component requirements and also to prove the absence of runtime error (no buffer overflows, no division by zero, etc.)

In our case, during the first run of GNATprove, we detected a possible overflow in an intermediate computation that we fixed with simple refactoring. It must be noted that this possible overflow might not have been detected with conventional XReq tests as these tests are mainly focused on features. Figure 9 illustrates the last execution report of the GNATProve tool on the “Battery_Manager” component, showing that all properties of interest are proved.

```
gnatprove -P/home/dev/CAP2018/pulsar-software/implementation/pulsar/battery_manager/battery_manager_impl.gpr -j0 -XPRJ_PLATFORM=hw-stub -XPRJ_RUNTIME=ravenscar-f
ress-bar -u pulsar-battery_manager.ads --report=all --level=4
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
pulsar-battery_manager.adb:23:110: info: overflow check proved
pulsar-battery_manager.ads:24:06: info: postcondition proved
pulsar-battery_manager.ads:50:04: info: initialization of "Battery_Manager.Battery_Failure" constituent of "Battery_Manager_State" proved
pulsar-battery_manager.ads:50:04: info: initialization of "Battery_Manager.Battery_Level_Sufficient_For_Flight" constituent of "Battery_Manager_State" proved
pulsar-battery_manager.ads:50:04: info: initialization of "Battery_Manager.Current_Cycles_Under_Low_Level" constituent of "Battery_Manager_State" proved
Summary logged in /home/dev/CAP2018/pulsar-software/implementation/pulsar/battery_manager/obj/gnatprove/gnatprove.out
[2017-10-31 16:01:11] process terminated successfully, elapsed time: 01.28s
```

Figure 9. Type 5 Ticket: GNATprove Report

Using formal methods techniques at this level allows us to completely skip writing and running XReq tests. The GNATprove tool takes about 30 seconds to analyse the “battery_manager”, “remote_control”, “notification_manager” and “inital_platform” packages. These packages include 12 component requirements. At this level, we must also take into account the necessary time to correct the errors reported by GNATprove. This time is variable depending on the complexity of errors.

Figure 10 shows the number of requirements (system, software and component) we implemented compared to the number of test cases. The figure also shows the total number of the produced tickets and their distribution by type.

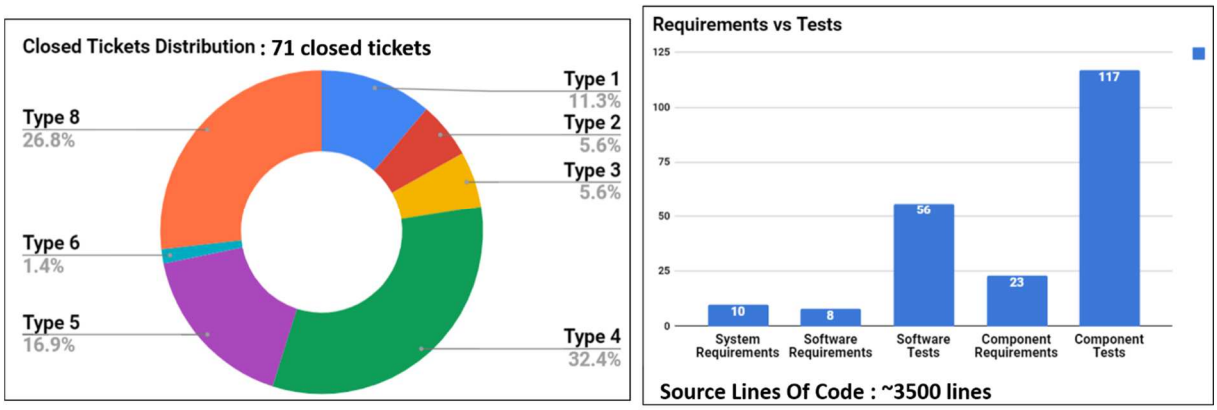

Figure 10. Development Advancement in Pulsar Flight System

C. Assessment of the Use of Formal Methods with SPARK

Using formal methods with SPARK in the development of Pulsar Flight System allowed us to derive useful insights, both at the technical and at the development process levels.

At the technical level, the major point to take into account is that, since the SPARK code to be analyzed must not contain pointers, it can be necessary and quite expensive to modify the software architecture of the impacted component in order to comply with this constraint. Thanks to the possibility to selectively activate or deactivate analysis on portions of the code, it is possible to use SPARK on isolated and specific code. In our study we had set aside the object-oriented concept for the component (in favor of an "Abstract Data Type" implementation concept), but we found during subsequent studies that it is possible to use object orientation with SPARK, at the cost of some adjustments.

At the development process level, it is more difficult initially to write SPARK contracts than to write natural language requirements for functional profiles, even if writing some ghost functions helps with the readability of SPARK. Writing such contracts requires a knowledge of the Ada / SPARK language that is not necessarily acquired by the person in charge of writing specifications. The advantage of using SPARK is that there is no need to write tests manually (with XReq), a task that is time-consuming and during which it can sometimes be difficult

to choose suitable test cases that cover the specification. The consistency verification of the code with the contract is ensured by formal verification, but the contract is assumed to be correct, which is difficult to assert at first sight.

V. Conclusion and Future Works

In conclusion, we found out that writing SPARK contracts does not allow us to write specifications easily, in the context of our development process. Indeed, during activities related to Type 4 tickets, writing SPARK contracts leads to writing code, which can be quite complex and requires from the editor, some knowledge of Ada 2012/SPARK 2014, knowledge not really necessary when writing requirements, even if they are LLR type. The traceability of such requirements is based on Ada comments mechanism, in the SPARK contract code. The verification of such contracts is relatively difficult, because if we are able to tell that the written code verifies the pre/post conditions with GNATprove tool, it becomes more difficult to say by reading the contract if it is compliant with the HLR requirements written during the activities of the Type 2 ticket. Finally, we realized that when SPARK contracts were set up, we have almost finally written the component code. In that way, we have merged the activities of tickets of Types 4 and 5, which compared to our development process cannot be relevant, having no more independence between the person writing the requirements and the tests specifications and the one in charge of implementing them. Nevertheless, the use of SPARK and formal methods allowed us to see potential runtime errors that we would not necessarily have detected with conventional XReq tests.

To better take advantages of the use of SPARK, especially in terms of saving time related to writing and validating XReq tests, we think that we could adjust our process to allow a more practical use of SPARK. During the activities of the Type 4 tickets, the explicit writing in natural language, via RMTool of the component requirements is saved, without writing any test. Then, the component requirement is translated into a SPARK contract, in another activity.

VI. References

[1] « Concept of Operations for Drones: A risk based approach to regulation on unmanned aircraft »: https://www.easa.europa.eu/document-library/general-publications/concept-operations-drones

[2] Pulsar Flight System: https://www.hionos.com/#pulsar

[3] SPARK: https://www.adacore.com/sparkpro

[4] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In Sixteenth International Symposium of Formal Methods, FM 2009, volume 5850 of Lecture Notes in Computer Science, pages 532–546, Eindhoven, The Netherlands, Nov. 2009. Springer-Verlag.

[5] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal Methods in Safety-Critical Railway Systems. In 10th Brasilian Symposium on Formal Methods, 2007.

[6] R. Jetley, S. P. Iyer, and P. L. Jones, "A Formal Methods Approach to Medical Device Review," IEEE Computer, vol. 39, pp. 61–67, 2006

[7] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A software analysis perspective ». Formal Asp. Comput., 27(3):573–609, 2015.

[8] Claude Marché and Yannick Moy, "The Jessie plugin for Deductive Verification in Frama-C." http://krakatoa.lri.fr/jessie.html, mars 2015. Consulté en juin 2015.

[9] CAP2018 - http://cap2018.minalogic.net/

[10] Ccrazyflie drone: https://www.bitcraze.io/crazyflie-2/

[11] Certiflie : https://github.com/AdaCore/Certyflie

[12] Nikolai Kosmatov, Claude Marché, Yannick Moy, Julien Signoles. « Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014". In Proc. of the 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2016), Corfu, Greece, October 2016, LNCS, vol. 9952, pages 461-478. Springer.

[13] Johannes Kanig, « A Comparison of SPARK with MISRA C and Frama-C." http://www.adacore.com/uploads/technical-papers/2016-10-SPARK-MisraC-FramaC.pdf

[14] Carl Brandon and Peter Chapin, "A SPARK/Ada CubeSat Control Program", Ada Europe 2013

[15] Carl Brandon and Peter Chapin, "The Use of SPARK in a Complex Spacecraft", HILT 2016

[16] Martin Becker, Emanuel Regnath, and Samarjit Chakraborty, "Development and Verification of a Flight Stack for a High-Altitude Glider in Ada/SPARK 2014", SAFECOMP 2017

[17] Emmanuel Chenu, "Agile & Lean software development for avionic software", ERTS 2012

[18] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman and Mike Hinchey, "Formal Versus Agile: Survival of the Fittest", Volume 42, Issue 9, Sept. 2009.

[19] CEOS project - https://www.ceos-systems.com/en/CEOS-Project-For-Pieces-Of-Work-Inspection-By Drones.html