Formal Verification With Frama-C: A Case Study in the Space Software Domain

Rovedy Aparecida Busquim e Silva, Nanci Naomi Arai, Luciana Akemi Burgareli, Jose Maria Parente de Oliveira, and Jorge Sousa Pinto

***Abstract*—With the increasing importance of software in the aerospace field, as evidenced by its growing size and complexity, a rigorous and reliable software verification and validation process must be applied to ensure conformance with the strict requirements of this software. Although important, traditional validation activities such as testing and simulation can only provide a partial verification of behavior in critical real-time software systems, and thus, formal verification is an alternative to complement these activities. Two useful formal software verification approaches are deductive verification and abstract interpretation, which analyze programs statically to identify defects. This paper explores abstract interpretation and deductive verification by employing Frama-C's value analysis and Jessie plug-ins to verify embedded aerospace control software. The results indicate that both approaches can be employed in a software verification process to make software more reliable.**

***Index Terms*—Aerospace safety, embedded software, formal verification, software quality, software safety.**

I. Introduction

FAILURES in safety critical real-time systems may cause injury, environmental damage, high financial losses, or even loss of life. As part of these systems, software has increasingly assumed important tasks. Surveys indicate that software in space systems has experienced an accelerated growth rate in recent decades [1] and that software has been involved in space software accidents [2]. The importance of software in a critical system failure can be demonstrated by the damage caused by the Ariane 5 launcher software. In June 1996, the first Ariane 5 flight ended in failure, resulting in self-destruction [3]. The complete loss of guidance and attitude information due to errors in the specification and design of the inertial reference software system was responsible for this failure. Specifically, the problem was a numerical error (overflow) in a floating-point number conversion. Other examples include the loss of the Mars Climate Orbiter in 1999 [4], the destruction of the Mars Polar Lander in the landing phase [5], and the loss of contact with SOHO (Solar Heliospheric Observatory) in 1998 [6]. Embedded software, such as that employed in space systems, is considerably more complex than IT, application, or desktop software due to the real-time and interface constraints [1]. Consequently, program correctness is a challenge in this area.

Currently, the general approach to check program correctness is through testing and simulation in a software verification and validation (V&V) process. Despite its importance, software testing is not yet a definitive solution for achieving predictability in real-time systems [7]. To complement these activities, aerospace standards recommend the use of formal methods [8], [9]. Formal software verification is an instance of applying a formal method during the development of software. Deductive verification, software model checking, and formal static analysis are some of the approaches that can be applied in formal software verification, particularly to prove program correctness at the implementation level.

Software model checking and formal static analysis based on abstract interpretation are attractive options because they do not require much effort from the user. However, problems such as state explosion and the generation of false alarms may occur. In contrast, deductive verification requires more user interaction to write the function contracts, but it has the advantage of not causing too many false alarms and termination issues [9]. Software model checking verifies arbitrary assertions; however, its abstraction techniques have not reached the same level of development as formal static analysis based on abstract interpretation. An alternative is to employ bounded model checking, but in this case, the analysis is only a partial set of program executions with a limited length [10].

This paper investigates the role of formal software verification in a V&V process with the goal of complementing the testing and simulation activities. We used formal static analysis based on abstract interpretation [11] and deductive verification [12]. The proposed approach applies these two mathematical formalisms to verify an embedded aerospace control software using the Framework for Modular Analysis of C (Frama-C) static analyzer [13]. The objective of this work is to obtain useful insights into the difficulties associated with applying this approach and to be able to analyze its practical feasibility as a formal activity in a software verification process.

The main contributions of this paper are as follows: 1) to disseminate our experience in applying the proposed approach not only to the scientific community but also to software engineering teams interested in verifying their software and 2) to discuss a case study from a real project in the aerospace field.

Manuscript received September 01, 2014; revised December 19, 2014, April 30, 2015, August 17, 2015, and November 23, 2015; accepted December 08, 2015. This work was supported in part by the Brazilian Space Agency under Grant 20VB. Associate Editor: W. E. Wong.

R. A. B. e Silva, N. N. Arai, and L. A. Burgareli are with the Division of Electronics, Institute of Aeronautics and Space, Sao Jose dos Campos, SP 12228-904, Brazil (e-mail: rovedyrabs@iae.cta.br; nancinna@iae.cta.br; lucianalab@iae.cta.br).

J. M. Parente de Oliveira is with the Division of Computer Science, Technological Institute of Aeronautics, Sao Jose dos Campos, SP 12228-900, Brazil (e-mail: parente@ita.br).

J. Sousa Pinto is with the HASLab/INESC TEC, University of Minho, Braga 4704-553, Portugal (e-mail: jsp@di.uminho.pt).

Digital Object Identifier 10.1109/TR.2015.2508559

The remainder of this paper is organized as follows. Section II describes the verification approach, including the tools and equipment used. Section III presents the related works. Section IV describes the case study and reports the practical experiments. Section V presents a discussion of the obtained results. Finally, Section VI presents the conclusion, contributions, and future work.

II. Formal Verification of Critical Software With the Frama-C Static Analyzer

We begin this section with an introduction to the Frama-C static analyzer, which is the tool on which our approach is based. We then describe the approach itself.

A. Tools and Equipment

Frama-C is an open-source platform dedicated to the static analysis of source code written in the C programming language, ISO C99 standard [14]. Despite the wide variety of uses for Frama-C, this work focuses on static analysis and deductive formal verification of critical software using the tool's value analysis and Jessie plug-ins.

The value analysis plug-in aims to statically analyze the code, computing variation domains for the program variables [13]. Additionally, this plug-in has the ability to detect errors that occur at run time and/or demonstrate their absence. It can be employed in the following tasks: familiarization with foreign code, automatic document production, bug detection, and guaranteeing the absence of bugs [15]. This plug-in implements a forward dataflow analysis based on the principles of abstract interpretation [13]. The plug-in performs a symbolic execution of the program, translating all operations into abstract semantics. Termination of looping constructs is ensured by widening operations [16]. For function calls, the plug-in performs a recursive inlining of the function, which ensures that the analysis is fully context sensitive [13]. To assist in the implementation of the analysis process, Frama-C provides libraries with predefined functions that can emulate the functions of the standard C language libraries, simulate the input parameters required for the plug-in, and enable printing of results during an analysis. The plug-in offers treatments related to loops, functions, conditional clauses (if-then-else), and disjoint intervals.

The Jessie plug-in allows users to perform deductive verification of C programs with annotations in the ANSI C Specification Language (ACSL) [13]. The deductive verification technique implemented in the Jessie plug-in automatically generates first-order logic proof obligations called verification conditions (VCs) using techniques such as Dijkstra's weakest precondition calculus and Hoare logic [12]. The generated formulas should then be shown to be valid, for instance, using an automated theorem prover (ATP). This calculus is used to convert C source code into an intermediate language by the Why platform [17] combined with the ACSL annotations [18]. In this work, we employed the ATPs Alt-Ergo [19], Simplify [20], CVC3 [21], Z3 [22], and Gappa [23]. The Jessie plug-in has some operating limitations: union types have limited support; casts between integers and pointers, as well as aliasing, require attention. The latest version of Frama-C provides two plug-ins for deductive verification: Jessie and WP. Generally, to execute deductive verification of C programs, there is a difficulty associated with the use of a memory model with the appropriate degree of accuracy. The Jessie plug-in works well with memory models that do not consider low-level properties. At the time when this work was conducted, the WP plug-in was still considered unstable, despite its ability to work with several memory models. Therefore, the Jessie plug-in was chosen to perform the deductive verification because of its stability.

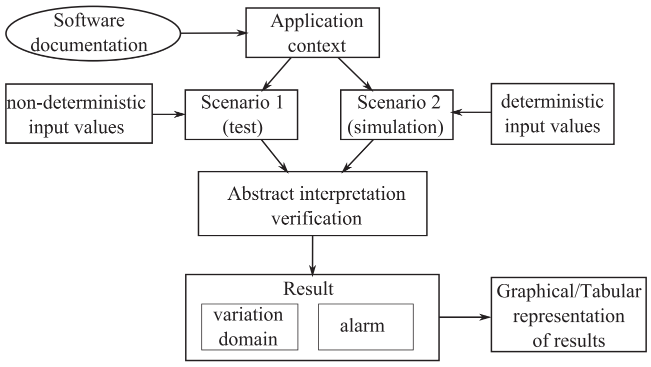

Software documentation
Application context
non-deterministic input values
Scenario 1 (test)
Scenario 2 (simulation)
deterministic input values
Abstract interpretation verification
Result
variation domain
alarm
Graphical/Tabular representation of results

Fig. 1. Overview of the approach for verification by abstract interpretation.

To conduct the formal software verification through abstract interpretation and deductive verification, we used the Frama-C Neon-20140301 release on a 2.9 GHz notebook with 8 GB of memory and 1 TB hard disk with the Yosemite operating system. To plot the graphs, we used a 2.20 GHz notebook with 2 GB of memory and a 130 GB hard disk with the Windows XP operating system.

B. Proposed Approach

The proposed approach employed two mathematical formalisms to perform the formal software verification based on the following steps:

1) use of formal static analysis by abstract interpretation based on a set of activities discussed in this work;
2) use of formal software verification by deductive verification based on a set of activities discussed in this work;
3) evaluation of the applicability of the proposed approach. A case study based on embedded software in an aerospace vehicle demonstrated the applicability of the approach. Because the data of the selected case study are classified, this works only presents a limited subset of the results.

We will now present details of the first two steps; the third step will be discussed in Section IV.

1) Formal Software Verification by Abstract Interpretation: An overview of the static analysis application based on abstraction using Frama-C's value analysis plug-in is shown in Fig. 1. Because the case study is critical real-time software that is responsible for acquiring data from sensors, processing such data, and controlling actuators, the proposed approach considered specific features, for example, functions for acquiring sensor data.

The approach began with preparing the application context, i.e., identifying the information required to perform the static analysis of the application. The software documentation could help in defining the application context. The approach defined two scenarios for performing the analysis: Scenario 1 and Scenario 2.

In Scenario 1, the plug-in was used for its primary purpose: to provide ranges of output values according to ranges of input values given by the user, as opposed to what a test-generation tool would typically do [15]. Test-generation tools work with a limited subset of test cases, whereas the plug-in works with a wider domain. This scenario is the most commonly employed by the users. The scenario context was based on non-deterministic input values for the application sensors. This means that the input values lied between the maximum and minimum limits of operation, obtained from a technical specification for the application.

In Scenario 2, the plug-in operated as a C interpreter, and the computed results were compared with the results produced by a compiled program [24]. A very useful characteristic of Frama-C's value analysis plug-in is that it can be used as an interpreter for C programs when a specific input data set is known in advance [25]. It suffices to run a sufficiently deep analysis: it will be completely deterministic and no false alarms will be generated, while undefined behaviors (not detectable by testing) will still be detected. In this scenario, the input values for the application sensors were obtained from the simulation results of the case study. The sensor values were deterministic, i.e., a specific value from the sensor was considered for each instant, and thus, Frama-C performed as a simulation tool.

The verification results included the variation domain for the application variables. The set inferred for the possible values of a variable allowed us to check whether the behavior of the implemented application was correct. In addition, alarms can be generated indicating possible run-time errors, such as occurrences of divisions by zero, invalid pointer access, overflow in signed arithmetic, and invalid function pointer access [15]. The results were recorded to an ASCII file for later analysis. Graphs and tables were used to facilitate the visualization and analysis of the data. In both scenarios, output data from previous simulations and/or analysis by system experts were used to check the correctness of the graphs and tables.

To detail the proposed approach, we constructed the activity diagram presented in Fig. 2. This approach is more than just executing a tool; it is a set of activities elaborated and organized in a systematic manner to perform the formal software verification. As shown in the diagram, we grouped the activities into two main phases: *context* definition and *implementation and refinement*.

The first phase consisted of preparing the application context for the case study through some definitions. It was necessary to select an entry point function for the analysis, identify the input data required to run the analysis, and choose the output data to be evaluated. The software documentation could support this task. Input data from sensors were provided in two ways: deterministic and non-deterministic.

The second phase encompassed the implementation and refinement of the analysis. The software functions to be analyzed were added incrementally. For each inserted function, inclusion treatment of the libraries, detection of the missing functions, handling alarms, and analysis of the variation domain were performed. This phase was an iterative and interactive process because the analysis required user intervention and was performed repeatedly until the intended result could be obtained. The user could adjust the accuracy of the analysis through the plug-in parameters taking into account the trade-off between efficiency and accuracy.

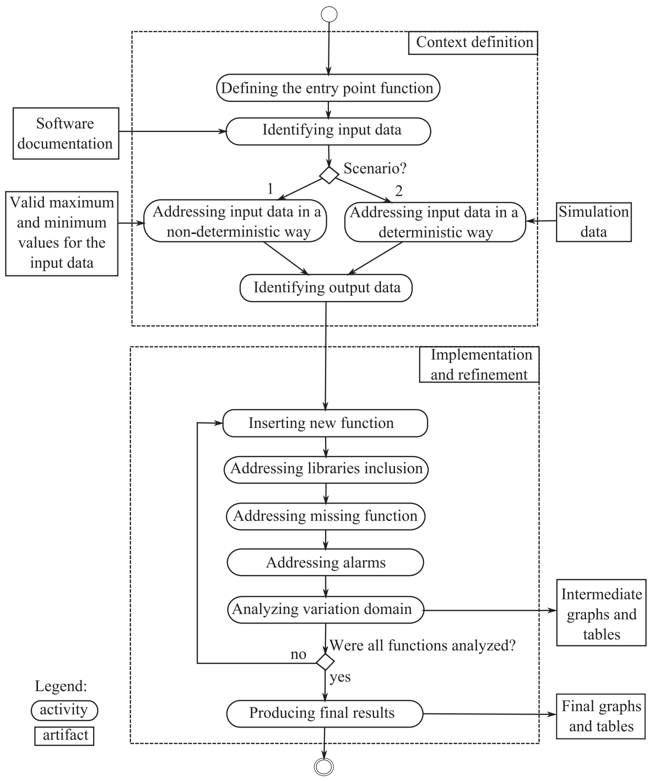

Context definition
Defining the entry point function
Software documentation
Identifying input data
Scenario?
1
2
Valid maximum and minimum values for the input data
Addressing input data in a non-deterministic way
Addressing input data in a deterministic way
Simulation data
Identifying output data
Implementation and refinement
Inserting new function
Addressing libraries inclusion
Addressing missing function
Addressing alarms
Analyzing variation domain
Intermediate graphs and tables
Were all functions analyzed?
no
yes
Producing final results
Final graphs and tables
Legend:
activity
artifact

Fig. 2. Activity diagram for executing the verification by abstract interpretation using Frama-C's value analysis plug-in.

Inclusion treatment of the libraries was performed because Frama-C's value analysis plug-in does not support certain external libraries, such as those related to the real-time operating system (RTOS). Furthermore, the plug-in only partially supports the mathematics library. Therefore, because keeping these libraries in the code implies a dependency chain that could not be provided, we decided to remove all *include* directives from the source code and then reinsert them incrementally according to the compiler error messages. Additionally, it was possible to detect unnecessary libraries.

Treatment of missing functions was related to the type of application submitted to the static analysis. For critical aerospace source code, missing functions could be categorized in the following types: hardware input/output (I/O), RTOS, mathematics, and other functions of the application itself. Table I shows the respective solutions to be provided to each function type.

At this point of the activities, the user was expected to evaluate and treat alarms. After this treatment, the variation domain for the variables needed to be evaluated, and the analysis continued until valid values for the observed magnitudes were obtained.

2) Formal Software Verification by Deductive Verification: An overview of deductive verification using Frama-C's Jessie plug-in is shown in Fig. 3. The approach began with a behavioral

TABLE I
MISSING FUNCTION CLASSIFICATION

Function type	Solution
Input hardware	Parameterizing the analysis by adding value intervals through tool functions
Output hardware	Observing intermediate results using the tool functions
RTOS	Providing an approximate implementation of the behavior
Mathematics	Providing an approximate implementation of the function and/or use the tool functions
Application	Code available

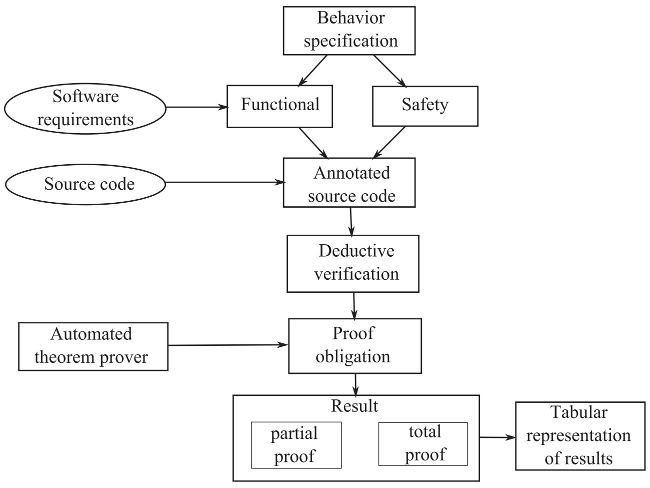


Fig. 3. Overview of the approach for deductive verification.

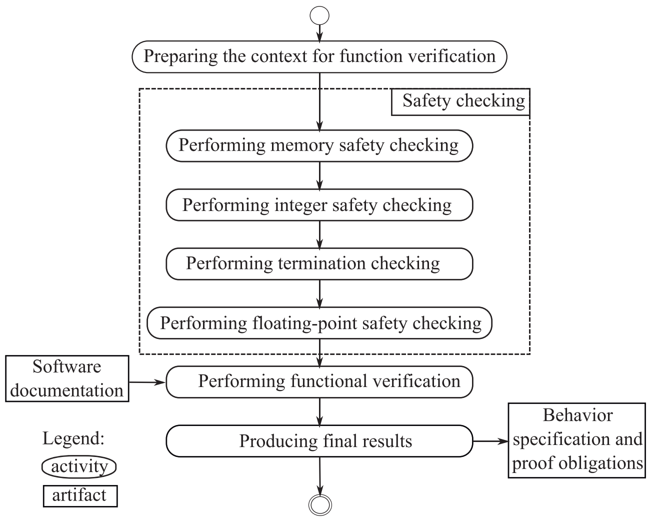

Preparing the context for function verification
Safety checking
Performing memory safety checking
Performing integer safety checking
Performing termination checking
Performing floating-point safety checking
Software documentation
Performing functional verification
Legend:
activity
artifact
Producing final results
Behavior specification and proof obligations

Fig. 4. Activity diagram for executing the deductive verification using Frama-C's Jessie plug-in.

specification of source code to be checked, which consisted of a safety and functional specification.

In this work, we used a safety specification to check for vulnerabilities in the source code. To identify possible source code vulnerabilities, we conducted a preliminary verification, and its results guided the safety specification writing, i.e., the code annotations. This safety check was associated with the following components [26]:

- memory safety deals with the validity of memory accesses to allocated memory;
- integer safety deals with the absence of integer overflow and the validity of operations with integers, such as absence of division by zero;
- floating-point safety deals with the absence of floating-point overflow;
- termination checks whether loops are always terminating, as well as recursive or mutually recursive functions.

The functional specification was directly related to the functional software requirements that could be obtained from the software documentation. In this approach, the main concept of functional verification was to prove the functions' postconditions when called in situations described by their preconditions.

Performing deductive verification produced proof obligations that shall (or not) be shown to be valid by ATPs. Ideally, all the proof obligations are shown to be valid. We investigated the proof obligations not shown to be valid to determine whether this is due to an application error or a tool limitation. The result of this investigation could guide the future use of the approach in actual software applications. The number of proof obligations checked, resulting in a total or partial proof of the source code, can be summarized in tables. The final result is a behavioral specification that is completely or partially proven.

To detail the proposed approach, we constructed the activity diagram presented in Fig. 4, which was performed for each function to be verified. Similar to verification by abstract interpretation, this approach consists of a set of activities elaborated and organized in a systematic manner to perform the verification by deductive verification.

The first activity was to prepare the context for checking the function and determining which library and application functions were required to perform this check. To aid this activity, we used Frama-C's Metrics plug-in. The use of this plug-in allowed for the automatic computation of various measures in the source code [27]. The syntactic metric used was the list of undefined functions.

The next four activities were related to the verification of source code safety checking: memory safety, integer safety, termination, and floating-point safety. In these activities, pragmas (clauses inserted in the source code to parameterize the type of safety check to be performed) permitted specific problems to be verified by steps. The pragmas employed in the safety check were *#pragma JessieIntegerModel(math)*, *#pragma JessieFloatModel(math)*, and *#pragma JessieTerminationPolicy(user)*.

The final activity was the functional verification. As previously mentioned, the first annotation inserted was the function's postcondition, extracted from the software documentation. The postcondition must be satisfied in situations or initial states satisfying the precondition. It is generally necessary to insert loop invariant annotations, assertions, and preconditions to prove the postcondition. The input/output parameters must be previously understood to identify the postcondition. The software documentation and Frama-C's Inout plug-in could help in this task.

The main result of these activities was the behavioral specification that is associated with the number of proof obligations checked, which characterize it as partial or total.

III. Related Works

The European Cooperation for Space Standardization standard ECSS-E-ST-40C on Space engineering Software (2009) contemplates a *software verification process*:

The software verification process is intended to confirm that adequate specifications and inputs exist for every activity and that the outputs of the activities are correct and consistent with the specifications and inputs.

This process is concurrent with all the previous processes.

The process consists of a significant number of activities, including the following:

1) verification of requirements baseline;
2) verification of the technical specification;
3) verification of the software architectural design;
4) verification of the software detailed design;
5) verification of code;
6) verification of software unit testing (plan and results);
7) schedulability analysis for real-time software.

Although the type of verification advocated in the standard is primarily based on testing, the use of formal tools is also contemplated. In particular, the standard recognizes that testing is not sufficient for verifying particularly difficult properties, e.g.:

The supplier shall verify source code robustness (e.g., resource sharing, division by zero, pointers, run-time errors).

AIM: use static analysis for the errors that are difficult to detect at run-time.

Other aspects of the verification are so demanding at the higher certification levels that the use of formal tools may be advisable. For instance, for coverage analysis, several published studies exist that show that the use of software model checking tools may be very helpful for increasing the productivity of this very demanding activity; see, for instance, [28]–[30].

Currently, there are a number of published works that show the application of formal methods in the verification of critical software in diverse domains. In particular, techniques based on abstract interpretation and deductive verification (which are used in the present paper) have now reached an industrial maturity level that allows them to be employed in addition to methods for partial exploration by classic testing. They have indeed been successfully applied in industrial projects in different domains. We will now consider examples from each of these domains.

Wiels *et al.* [31] provide an overview of Onera's research in applying formal verification at model and code levels in the context of aerospace software. The research explores the specifics of applying formal methods to aerospace, model-driven engineering at the platform level, cooperation of analysis techniques, and test automation using formal methods.

In the avionics domain, companies such as Dassault-Aviation and Airbus have successfully applied formal verification early on as a cost-effective alternative to testing [9]. They follow the supplement on formal methods called DO-333, part of the DO178C standard. Airbus verifies functional properties through “unit proof” (the term “unit proof” echoes the name of the classical technique it replaces: unit testing). This activity has replaced some of the testing activities at Airbus for parts of critical embedded software on the A400M military aircraft and the A380 and A350 commercial aircraft. Dassault-Aviation has used formal verification techniques experimentally to replace integration robustness testing. The source code is automatically generated from a graphical model. The model contains a set of assertions that are expected to be met in both normal and abnormal input conditions for the model to operate properly. These assertions are translated into the source code and verified using Frama-C.

In the railway domain, the results produced by applying formal verification using Frama-C are compared with traditional unit testing [32]. The software requirements are written in ACSL annotations and inserted into the source code to check whether the implementation is correct with respect to its specification. The work concludes that the process of formal specification helps to detect inaccuracies and ambiguities in the requirements and suggests that it is necessary to discuss the ways in which formal methods may replace certain test activities in this domain.

The literature review also demonstrates the viability of employing deductive verification in a number of industrial case studies [33]–[35]. In addition to program correctness, deductive verification can be used for other purposes. The annotations inserted in the source code can be used to detect and diagnose software bugs and to describe contracts between functions [36].

One of the most widely used frameworks for the development of high-assurance software (in terms of its functional and nonfunctional properties) is the SPARK programming language and toolset [37]. The language itself is based on a restricted subset of the Ada programming language (adequate for the development of safety critical software), complemented by annotations that can be used in particular to write contracts describing aspects of the system's properties. The SPARK platform provides a set of verification tools for reasoning about the correctness of the source code, which makes it possible to detect problems early in the software life cycle. The toolset contains both automatic and interactive tools that can check (statically) for the absence of run-time errors and for functional correctness (based on deductive verification).[1]

Formal static analysis, based on abstract interpretation, is a type of formal software verification that is capable of providing

[1]An up-to-date list of industrial projects using SPARK can be found at http://www.adacore.com/sparkpro/projects.

valid results for all input values proposed by the user [15]. It extracts faults detected at run time while requiring little assistance from the user, in addition to having good scalability at a limited accuracy cost [38], [39]. These faults may be false positive, but never false negative, which defines formal static analysis based on abstract interpretation as conservative. Its application to software verification in the aerospace field shows that the performance of static analysis tools has significantly improved [40]. The cited works [38]–[40] indicate the viability and efficiency of using formal static analysis based on abstract interpretation in a software V&V process for an aerospace application.

Note that there have been considerable advances in automated approaches to assertion checking (as an alternative to deductive verification techniques), building on one of the following techniques to attain a manageable state space to be explored:

- bounded verification techniques (a limited exploration of the state space): this is in general always helpful for bug finding and may in many cases be sound and complete because a substantial amount of critical code relies on bounded iteration only.
- existentially abstract models: models of the system that group many different concrete states into a single abstract one in a way that preserves safety (reachability) violations at the cost of introducing spurious safety-violating executions. The abstract model is typically constructed iteratively by starting with a very abstract approximation that is progressively refined based on the spurious counterexamples found (a process known as counterexample-guided abstraction refinement [41]).

Examples of tools based on these techniques include CBMC [42], the flagship bounded model checker for C programs, and abstraction tools such as SLAM [43], the CEGAR-based MS Windows device driver verifier, and TASS [44], a suite that uses symbolic execution and model checking techniques to verify safety properties.

The present paper explores the application of abstract interpretation and deductive verification, employing Frama-C's value analysis and Jessie plug-ins, to verify embedded aerospace control software. In the software verification process, the activity of applying formal verification is a difficult task, and one reason for this is the difficulty in learning and applying the tools that implement such techniques. Due to the very limited availability of detailed examples of how to use Frama-C, we elaborated a set of activities organized in simple flowcharts. In this way, we expect to facilitate the Frama-C usage and its integration in the software development process.

IV. Experiments

We will begin this section by introducing the case study, and we will then present the experiments and obtained results for both the abstract interpretation and deductive verification activities of our approach.

A. Case Study

The selected case study is the on-board software in the Brazilian Satellite Launch Vehicle (VLS-V03) developed at the Institute of Aeronautics and Space (IAE), an organization subordinated to the Department of Science and Aerospace Technology (DCTA) of the Brazilian Air Force (FAB). VLS is a controlled vehicle with four stages capable of launching satellites weighing 100 to 350 kg to altitudes of up to 1000 km. Three prototypes have been developed: VLS-V01, VLS-V02, and VLS-V03. The first launch occurred in 1997 with a failure in the first stage igniter; the second launch occurred in 1999 with a failure in the second stage booster; and in 2003, there was an accident in the preparation of the third prototype due to a failure in a mechanical safety device in the first stage [45].

The embedded software is critical real-time software that is responsible for navigation and guidance of the vehicle, control over the actuators, management of the events sequence, transmission of telemetry data, and vehicle pre-launch activities [46]. For simplification, our case study is restricted to the flight control of the vehicle first stage, with the execution of navigation, attitude control, and event sequence algorithms.

This case study was selected because of its high criticality and complexity. Its characteristics allow Frama-C's value analysis and Jessie plug-ins to be used. Frama-C's value analysis plug-in is applicable to embedded codes due to their characteristics such as the absence of recursion and dynamic memory allocation and no calls to external libraries (except the RTOS) [15]. Frama-C's Jessie plug-in is appropriate for applications written in the C programming language because this language has safety vulnerabilities, such as buffer overflow. Moreover, this plug-in is appropriate for algorithms with many numerical calculations that should be checked carefully because they are more prone to errors because they utilize floating-point arithmetic. The analyzed source code consisted of sixteen functions from the case study. In addition to these function, another thirteen functions were added to the context definition such that the verification by abstract interpretation could be performed.

B. Abstract Interpretation

The final results of this verification are presented according to each scenario. We generated nine graphs for Scenario 1 and twenty-three graphs for Scenario 2. However, because classified data were used, only graphs related to the position and inertial linear velocity of the vehicle in orbit are presented.

1) Results From Scenario 1: We compared the Frama-C output to the hybrid simulation output obtained by the control system team. The hybrid simulation is a hardware-in-loop simulation that simulates the complete system, that is, the on-board computer with the control laws, the vehicle dynamics, the actuator dynamics, and the sensor dynamics, to assess the performance of a satellite launcher control system [47].

Table II shows the intervals computed by Frama-C's value analysis plug-in, as well as the minimum and maximum values of two navigation algorithm variables extracted from the hybrid simulation profile.

The graphs for Scenario 1 show the variation domain for the output variables of the navigation and attitude control algorithms. To determine whether the computed variables were valid, we plotted the data from the hybrid simulation in the same graph. Figs. 5 and 6 show the maximum and minimum values of the intervals, at each time instant for the vehicle position data on the X axis and the inertial linear velocity data on the Z axis,

TABLE II
MAXIMUM AND MINIMUM INTERVALS GENERATED BY THE VALUE ANALYSIS

Variable	Frama-C		Simulation	
	Min	Max	Min	Max
Voo_Nav_Vel_Inerc_Vz	0.44282400	1.574144959	0.446638	0.852687
Voo_Nav_Pos_Inerc_X	6373.05810	6439.161133	6378.307443	6399.182104

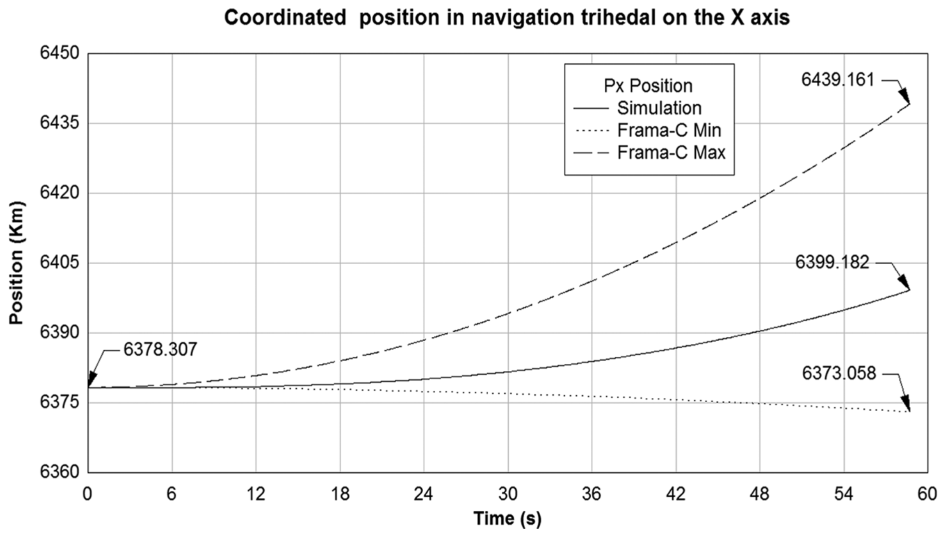


Fig. 5. Graph of X position in Scenario 1.

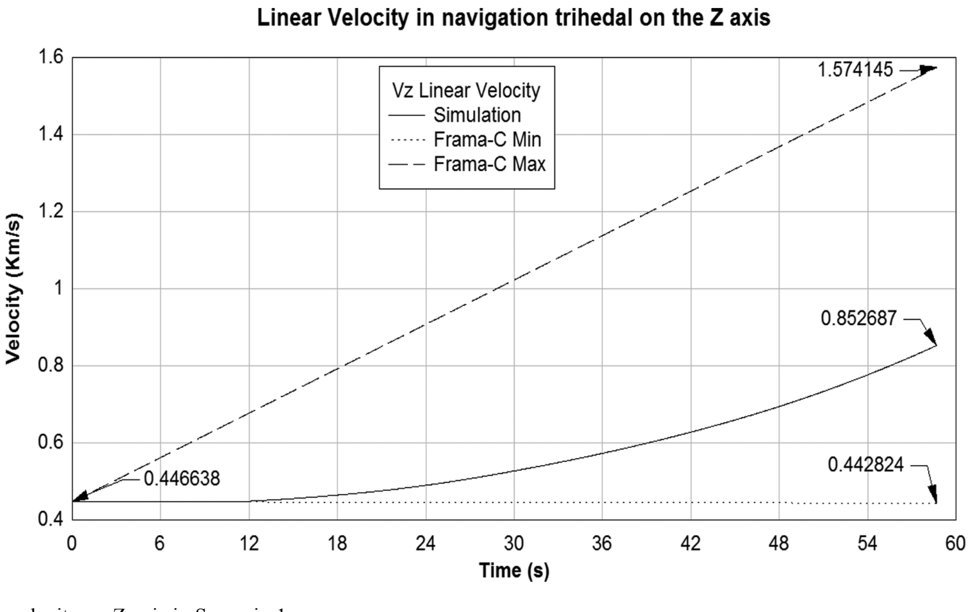


Fig. 6. Graph of linear velocity on Z axis in Scenario 1.

as computed by Frama-C's value analysis plug-in. We compared these data to the profile of the same variables from the hybrid simulation. The use of non-determinism in the analysis implied that the range between the curves of minimum and maximum data produced by Frama-C increased over time. A discussion about this result is presented in Section V-A1.

2) Results From Scenario 2: Frama-C's value analysis plug-in produced variable profiles that were very similar to the hybrid simulation, and in most cases, the curves were so similar that they are visually coincident. Figs. 7 and 8 show the position data on the X axis, and Figs. 9 and 10 show the inertial linear velocity data on the Z axis at each instant of time, as computed by Frama-C's value analysis plug-in. We compared these data to the results of the same variables from the hybrid simulation. The behavior of Frama-C as an interpreter produced results that were coincident with those produced by the simulation. A discussion of these results is presented in Section V-A2.

3) Final Results: The anomalies detected by the verification are related to two software products—documentation and implementation—and can be observed in Table III.

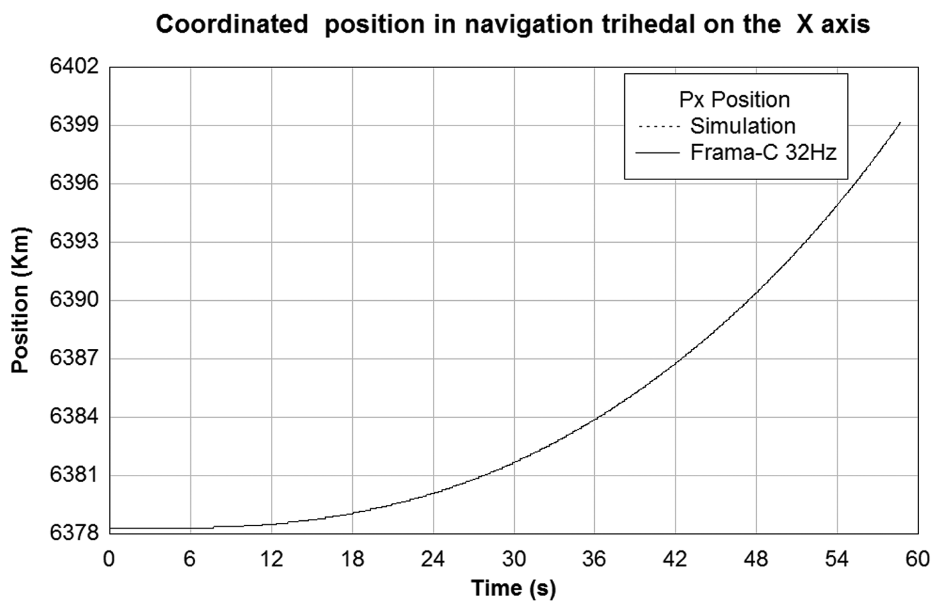


Fig. 7. Graph of X position in Scenario 2.

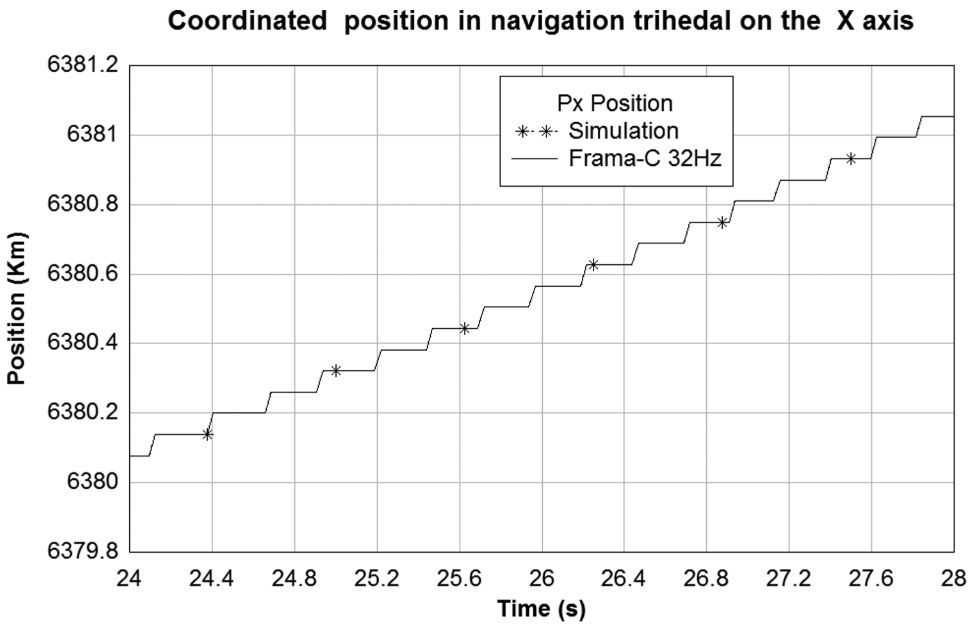


Fig. 8. Graph of X position in Scenario 2 with zoom.

4) Performance and Size: We optimized the analysis execution through the *-no-result* option applied in some functions to avoid logging unnecessary variables such that the size of the generated output file would not become too large. Table IV shows the output file size and processing time obtained using the Unix time command.

C. Deductive Verification

1) Proof Obligations: The final result from the verification is presented in Table V, which shows the number of proven and unproven proof obligations. The unproven VCs presented are related to the following:

- bitwise operations that are not supported by the plug-in;
- floating-point arithmetic that are not mostly proven by ATPs;
- postcondition and precondition for call.

In program correctness, some types of VCs cannot be shown to be valid for ATPs, which is a consequence of using first-order logic. In this case, the solution is to use interactive provers with manually written proofs.

2) Annotations: The number of annotations included in the source code of the case study is categorized in accordance with Table VI. The number of annotations provided an estimation of the computational effort required to perform the verification.

As an example, not meant to instruct the reader on the use of Frama-C or Jessie, Fig. 11 shows the function1 source code with annotations. This function is responsible for reading the

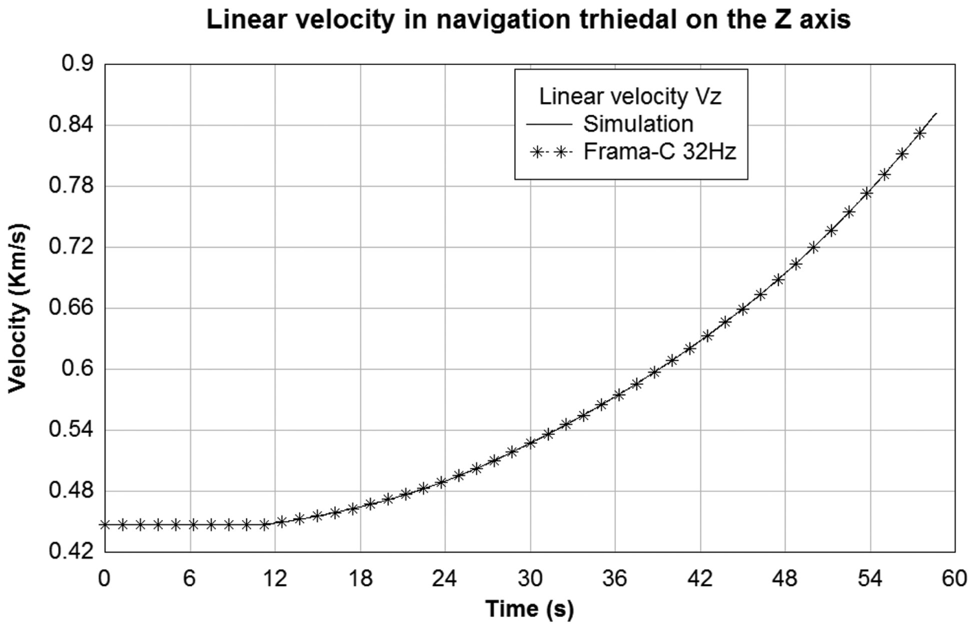


Fig. 9. Graph of linear velocity on Z axis in Scenario 2.

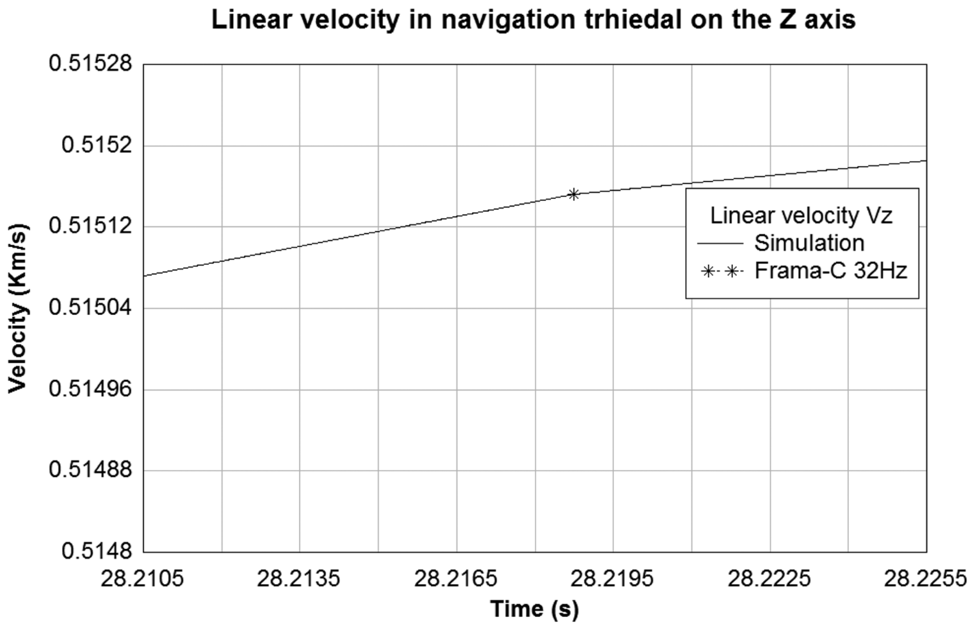


Fig. 10. Graph of linear velocity on Z axis in Scenario 2 with zoom.

flight-events table, searching for the time of their occurrence. To insert the annotations in the code, we first ran the Metrics plug-in. This plug-in did not indicate any occurrences of function calls; thus, in this case, inserting contracts in called functions was not required. In addition, the plug-in indicated the existence of one loop that had to be addressed with annotations. Memory safety was the first check to be performed. It was executed without annotations, with the relevant pragmas as discussed in Section II.B2, and it generated one pointer dereference VC that was proven by inserting an invariant loop annotation (line 9 in Fig. 11). We addressed the loop inserting annotations to identify variables modified in the loop (line 12 in Fig. 11) and to identify properties already ensured by the loop (lines 10–11 in Fig. 11). The second check, integer safety, computed arithmetic overflow VCs that were successfully proven. The third check, termination, generated a loop termination VC that was proven by inserting a loop variant decrease annotation (line 13 in Fig. 11). The fourth check, floating-point safety, did not produce any VCs. Finally, after these checks, the Inout plug-in was executed, and its result showed the assigned variables. We used this result and the information obtained from the software documentation to insert the final annotations (lines 1–3 in Fig. 11) and to execute the functional verification. Fig. 12 shows the graphic interface of the Why Platform. The screenshot shows the installed ATPs (on the left top side), the VCs (on the center), the source code translated into the intermediate language (on the

TABLE III
ANOMALIES DETECTED FROM ABSTRACT INTERPRETATION VERIFICATION

Software product	Anomaly
Specification Data Dictionary	Wrong variation domain of angular velocity in roll axis variable
Software Design Document	Misdescription of a sensor angular velocity component in the navigation module specification
Design Data Dictionary	Misdescription of a sensor variable in the definition type
Source code (ten functions)	Thirty unnecessary/duplicate inclusions of RTOS, mathematics, and application libraries
Sensor data acquisition function	Global variable passed by value detected, also by a manual code inspection [48]
Attitude control function	Dead code detected
Navigation function	Incorrect comment related to the sensor angular velocity component

TABLE IV
PROCESSING TIME AND OUTPUT FILE SIZE

	Scenario 1	Scenario 2
Performance	181m48.937s	6m42.083s
Output file size	113.390 MBytes	87.171 MBytes

TABLE V
PROOF OBLIGATIONS

Source Code	Proven	Unproven	Total
function0	19	-	19
function1	16	-	16
function2	70	-	70
function3	153	-	153
function4	2645	-	2645
function5	4	-	4
function6	1801	113	1914
function7	9	-	9
function8	100	5	105
function8a	8	-	8
function8b	152	-	152
function8c	386	-	386
function8d	177	-	177
function8e	255	118	373
function8f	20	4	24
function9	23	-	23
function10	21	-	21
function11	44	7	51

TABLE VI
ANNOTATIONS TO THE APPLICATION

Type of annotation	Number of annotations
assert	202
assigns	60
axiomatic	2
ensures	202
ghost variable	56
lemma	2
loop assigns	2
loop invariant	18
loop variant	9
requires	297
&& operator	311
|| operator	18
⇒ operator	141

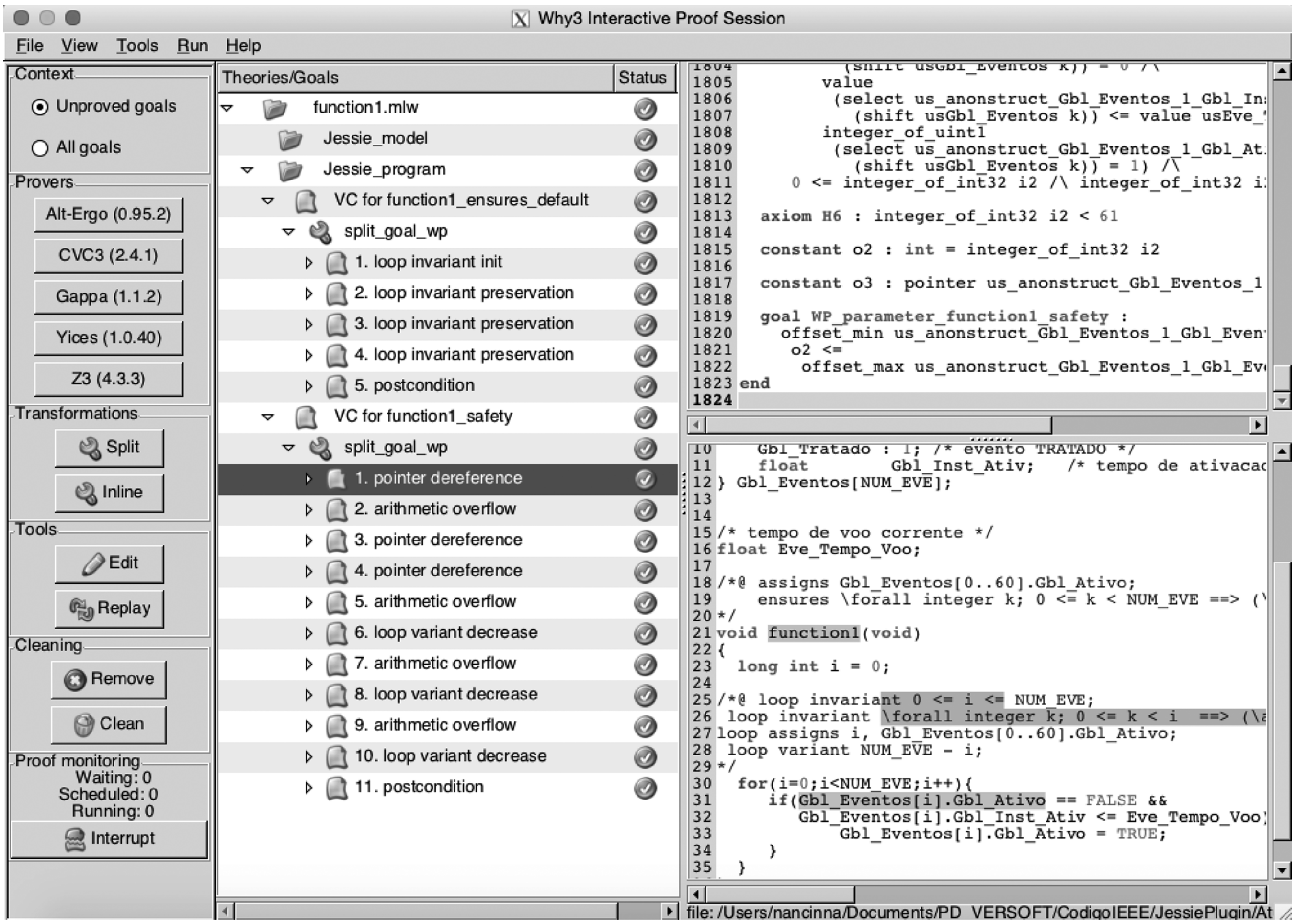
right top side), and the goals and hypotheses (on the right bottom side). The status column indicates that all the goals were proven through the tick icons.

We provide another example to illustrate the occurrence of unproven VCs. Fig. 13 shows the function11 source code with annotations. This function acts on the analog I/O of the *Programmed Input/Output* (PIO) boards. First, it was necessary to change the static variable *Pio_Data* to a global variable [49]. Next, we ran the Metrics plug-in, which did not indicate the existence of any loops or pointer dereferences. However, the plug-in identified the existence of an undefined function that corresponds to an RTOS function that writes a byte to an I/O address [50]. In this case, it was necessary to provide the function definition and its contract, as shown in Fig. 14. The memory safety check was executed with the relevant pragmas following Section II.B2 and the sysOutByte function contract, and it generated two unproven pointer dereference VCs and four unproven precondition for call VCs. After analyzing the VCs, we identified that the former are related to the array index passed as a parameter to the function that should be in a valid range. To prove these VCs, we inserted annotations (line 9 in Fig. 13). The latter are related to the sysOutByte() function that is called with a char in the second parameter, whereas an unsigned char was expected. This is probably an anomaly in the source code implementation and was reported to the development team. Next, an integer safety check was performed and generated seven unproven arithmetic overflow VCs. We inserted annotations (lines 10–12 in Fig. 13) to guarantee valid value ranges to the bitwise operations. However, we verified

```
/*@ assigns Gbl_Eventos[0..60].Gbl_Ativo;
  @ ensures \forall integer k;  <= k < NUM_EVE ==> (\old(Gbl_Eventos[k].Gbl_Ativo) == FALSE
  @ &&\old(Gbl_Eventos[k].Gbl_Inst_Ativ)<= Eve_Tempo_Voo ==> Gbl_Eventos[k].Gbl_Ativo == TRUE);
*/
void function1(void)
{
long int i = 0;

/*@ loop invariant 0 <= i <= NUM_EVE;
  @ loop invariant \forall integer k; 0 <= k < i  ==> (\at(Gbl_Eventos[k].Gbl_Ativo, Pre) == FALSE
  @ && \at(Gbl_Eventos[k].Gbl_Inst_Ativ, Pre) <= Eve_Tempo_Voo ==> Gbl_Eventos[k].Gbl_Ativo == TRUE);
  @ loop assigns i, Gbl_Eventos[0..60].Gbl_Ativo;
  @ loop variant NUM_EVE - i;
 */
  for(i=0;i<NUM_EVE;i++){
  if(Gbl_Eventos[i].Gbl_Ativo == FALSE && Gbl_Eventos[i].Gbl_Inst_Ativ <= Eve_Tempo_Voo) {
           Gbl_Eventos[i].Gbl_Ativo = TRUE;
     }
  }
}
```

Fig. 11. ACSL annotations written to function1.



```
function1.mlw
  Jessie_model
  Jessie_program
    VC for function1_ensures_default
      split_goal_wp
        1. loop invariant init
        2. loop invariant preservation
        3. loop invariant preservation
        4. loop invariant preservation
        5. postcondition
    VC for function1_safety
      split_goal_wp
        1. pointer dereference
        2. arithmetic overflow
        3. pointer dereference
        4. pointer dereference
        5. arithmetic overflow
        6. loop variant decrease
        7. arithmetic overflow
        8. loop variant decrease
        9. arithmetic overflow
        10. loop variant decrease
        11. postcondition
```

```
          value
           (select us_anonstruct_Gbl_Eventos_1_Gbl_In
             (shift usGbl_Eventos k)) <= value usEve_
          integer_of_uint1
           (select us_anonstruct_Gbl_Eventos_1_Gbl_At
             (shift usGbl_Eventos k)) = 1) /\
      0 <= integer_of_int32 i2 /\ integer_of_int32 i

  axiom H6 : integer_of_int32 i2 < 61

  constant o2 : int = integer_of_int32 i2

  constant o3 : pointer us_anonstruct_Gbl_Eventos_1

  goal WP_parameter_function1_safety :
    offset_min us_anonstruct_Gbl_Eventos_1_Gbl_Even
     o2 <=
      offset_max us_anonstruct_Gbl_Eventos_1_Gbl_Ev
end
```

```
    Gbl_Tratado : 1; /* evento TRATADO */
    float         Gbl_Inst_Ativ;   /* tempo de ativacao
} Gbl_Eventos[NUM_EVE];


/* tempo de voo corrente */
float Eve_Tempo_Voo;

/*@ assigns Gbl_Eventos[0..60].Gbl_Ativo;
    ensures \forall integer k; 0 <= k < NUM_EVE ==> (\
*/
void function1(void)
{
  long int i = 0;

/*@ loop invariant 0 <= i <= NUM_EVE;
 loop invariant \forall integer k; 0 <= k < i  ==> (\a
loop assigns i, Gbl_Eventos[0..60].Gbl_Ativo;
 loop variant NUM_EVE - i;
*/
  for(i=0;i<NUM_EVE;i++){
     if(Gbl_Eventos[i].Gbl_Ativo == FALSE &&
        Gbl_Eventos[i].Gbl_Inst_Ativ <= Eve_Tempo_Voo)
            Gbl_Eventos[i].Gbl_Ativo = TRUE;
     }
  }
```

file: /Users/nancinna/Documents/PD VERSOFT/CodigoIEEE/JessiePlugin/At

Fig. 12. Screenshot of function1 in the Why Interactive Proof Session tool.

that the ATPs did not prove the VCs. Researching this issue, we determined that this is probably related to a problem that has been reported regarding the bitwise operation [51]. The termination and floating-point safety checks did not generate any VCs. Finally, in the functional checks, the Inout plug-in showed the assigned variables. We wrote the final annotations based on this result and the software documentation (lines 13–14 in Fig. 13). Fig. 15 shows this example in the graphic interface of the Why platform.

3) Final Results: The anomalies detected by the verification are related to two software products—documentation and implementation—and can be observed in Table VII.

V. Discussion

A. Abstract Interpretation

1) Scenario 1: The execution of the verification did not emit any alarms. Based on the analysis of the results, we did not iden-

```
/*@ requires address >= 0x81600000 && address <= (0x81700000 + 3);
    requires 0 <= data && data  <= 255;
    assigns \nothing;
*/
 void  sysOutByte(unsigned long int address, unsigned char data);

 char Pio_Data[2][4] = {{0,0,0,0}, {0,0,0,0}};

/*@ requires 0<=Id_Board<2 && 0<=Id_Port<4;
     requires Inc_Data == 1 || Inc_Data == 3 ||Inc_Data == 4 ||Inc_Data == 6 ||Inc_Data == 8 ||Inc_Data == 24
              ||Inc_Data == 32 || Inc_Data == 64;
     requires \forall integer k; 0 <= k < 2 ==> (\forall integer j; 0 <= j < 4 ==> -128<=Pio_Data[k][j]<=127);
     assigns Pio_Data[Id_Board][Id_Port];
     ensures \forall integer k; 0 <= k < 2 ==> (\forall integer j; 0 <= j < 4 ==> -128<=Pio_Data[k][j]<=127);
*/
void function11(char Id_Board, char Id_Port, char Inc_Data, char Id_Action)
{
  char Data_Temp;
  unsigned long int Address;

   if(Id_Action == WRITE_0) {  Data_Temp = Pio_Data[Id_Board][Id_Port] & (char)(~ Inc_Data); }
                      else{  Data_Temp = Pio_Data[Id_Board][Id_Port] | (char)Inc_Data;     }

   Pio_Data[Id_Board][Id_Port] = Data_Temp;

   if(Id_Board == PIO1) {   Address = ADDR_PIO1 + (int)Id_Port; }
                  else{   Address = ADDR_PIO2 + (int)Id_Port; }

  sysOutByte(Address, Data_Temp);
}
```

Fig. 13. ACSL annotations written to function11.

TABLE VII
ANOMALIES DETECTED FROM DEDUCTIVE VERIFICATION

Software product	Anomaly
Software Design Document	One missing output variable in function11 structure chart
	Two incorrect parameters in function8f structure chart
Design Data Dictionary	One incorrect variable type definition in function11
	Name and type inconsistency of two parameters in the dictionary and source code
	Type definition inconsistency of three variables in the dictionary and source code
Function to acting on the analog I/O of the PIO boards	One missing typecast in the parameter passed to I/O function, also detected by a manual code inspection [48]
	One argument passed as parameter exceeds the parameter type size in the prototype of the function
Reference event treatment function	Redundancy of code in *if* statement
	Inconsistency in one variable initiation and its type
Function to limit the deflection on mobile nozzles	Incorrect comments related to input and output variables names
Navigation function	Dead code detected

```
/*@ requires address >= ADDR_PIO1 && address <= (ADDR_PIO2 + 3);
  @ requires 0 <= data && data <= 255;
  @ assigns \nothing;
*/
void sysOutByte(unsigned long int address, unsigned char data);
```

Fig. 14. ACSL annotations written to the sysOutByte() function.

tify defects such as buffer overflows in the application source code.

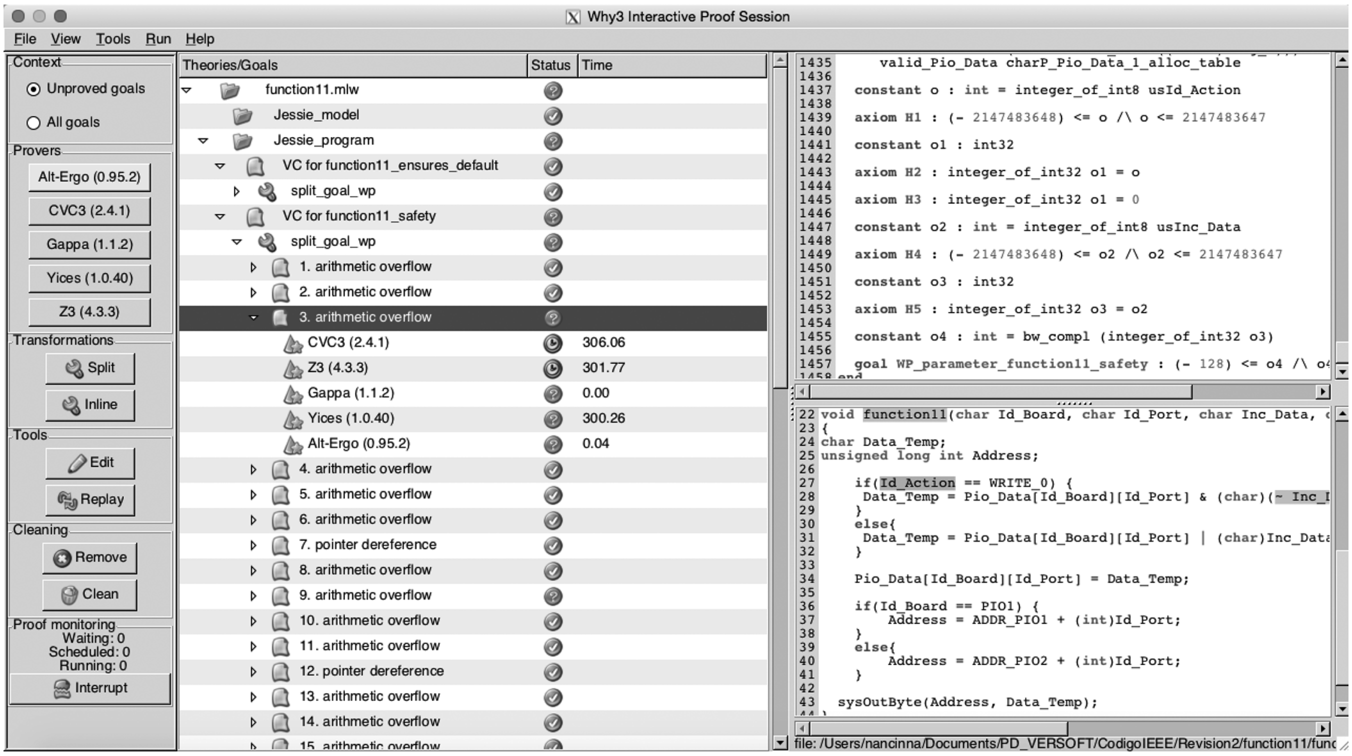
The analysis provided valid domains for all variables, i.e., Frama-C computed variation domains that satisfy the (minimum and) maximum value allowed by the variable type. Table II and the graphs presented in Figs. 5 and 6 show some selected variables and their analysis results. Fig. 16 shows the bar graph of the coordinated position in navigation trihedral on the Y axis. We selected this type of graph to more clearly illustrate that the intervals of the variable computed by Frama-C contain the simulation values, i.e., there is no value of the *Py* variable from the simulation that is below the minimum limit or above the maximum limit of Frama-C.

Because simulation data are used as a reference to prove the algorithms correctness of the on-board software, we use these data as benchmarks for the Frama-C output. If the Frama-C bounds do not contain the simulation results, then the correctness of the source code of the case study could not be ensured. In this case, the source code and Frama-C outputs would have to be analyzed and possibly more experiments performed.

In summary, the analyzed results from Frama-C's value analysis plug-in indicate that the algorithms are correctly implemented without problems such as division by zero, invalid pointer access, buffer overflows, and other run-time errors.

2) Scenario 2: Analysis of the results verified that the execution of the analysis occurred as expected, compared with a simulation of a real application execution.

As a result of the Scenario 2 analysis, three aspects are discussed: the comparison of data from the hybrid simulation



```
Theories/Goals                               Status  Time
function11.mlw
  Jessie_model
  Jessie_program
    VC for function11_ensures_default
      split_goal_wp
    VC for function11_safety
      split_goal_wp
        1. arithmetic overflow
        2. arithmetic overflow
        3. arithmetic overflow
          CVC3 (2.4.1)                               306.06
          Z3 (4.3.3)                                 301.77
          Gappa (1.1.2)                              0.00
          Yices (1.0.40)                             300.26
          Alt-Ergo (0.95.2)                          0.04
        4. arithmetic overflow
        5. arithmetic overflow
        6. arithmetic overflow
        7. pointer dereference
        8. arithmetic overflow
        9. arithmetic overflow
        10. arithmetic overflow
        11. arithmetic overflow
        12. pointer dereference
        13. arithmetic overflow
        14. arithmetic overflow
        15. arithmetic overflow
```

```
    valid_Pio_Data charP_Pio_Data_1_alloc_table

  constant o : int = integer_of_int8 usId_Action

  axiom H1 : (- 2147483648) <= o /\ o <= 2147483647

  constant o1 : int32

  axiom H2 : integer_of_int32 o1 = o

  axiom H3 : integer_of_int32 o1 = 0

  constant o2 : int = integer_of_int8 usInc_Data

  axiom H4 : (- 2147483648) <= o2 /\ o2 <= 2147483647

  constant o3 : int32

  axiom H5 : integer_of_int32 o3 = o2

  constant o4 : int = bw_compl (integer_of_int32 o3)

  goal WP_parameter_function11_safety : (- 128) <= o4 /\ o
end
```

```
void function11(char Id_Board, char Id_Port, char Inc_Data, c
{
char Data_Temp;
unsigned long int Address;

    if(Id_Action == WRITE_0) {
     Data_Temp = Pio_Data[Id_Board][Id_Port] & (char)(~ Inc_I
    }
    else{
     Data_Temp = Pio_Data[Id_Board][Id_Port] | (char)Inc_Data
    }

    Pio_Data[Id_Board][Id_Port] = Data_Temp;

    if(Id_Board == PIO1) {
        Address = ADDR_PIO1 + (int)Id_Port;
    }
    else{
        Address = ADDR_PIO2 + (int)Id_Port;
    }

  sysOutByte(Address, Data_Temp);
```

file: /Users/nancinna/Documents/PD_VERSOFT/CodigoIEEE/Revision2/function11/func

Fig. 15. Screenshot of function11 in the Why Interactive Proof Session tool.

versus data produced by Frama-C, the data accuracy, and discrepant curves.

We used the hybrid simulation curves to determine whether the curves obtained by Frama-Cs value analysis plug-in were correct. The hybrid simulation curves are generated from the data that the on-board software sends to the telemetry system.

The on-board software generates data at a 64 Hz sampling rate. However, part of these data are sent to the telemetry system at 32 Hz due to the system requirements. In addition, the sent data are also truncated and less precise because of the formatting process applied to the data due to the characteristics of the data bus used. Thus, to be able to more accurately compare Frama-C and the hybrid simulation curves, we included the function that formats the data to be sent to the telemetry system into the source code under Frama-C analysis. Additionally, sampling data reduction at a frequency of 32 Hz was required. Figs. 7, 8, 9, and 10 in Section IV-B2 show that Frama-C and the hybrid simulation curves are similar, as expected.

Because Frama-C does not have transmission frequency or data formatting restrictions, it was possible to conduct the analysis with Frama-C independently of these restrictions. Fig. 17 shows the linear velocity at a 64 Hz sampling rate compared with equivalent data from the simulation at a 32 Hz sampling rate. Consequently, Frama-C generates data that contain more information, accuracy, and similarity with the real flight data, which assists in analyzing algorithms in development and/or in the interpreting the post-flight data.

Exclusively in the graphs of the acting of two motors of the vehicle (not shown here because it is classified data), there was a significant difference between the curves of the hybrid simulation and Frama-C. To determine the cause of this discrepancy, the analysis context was checked. After extensive inspection, we did not detect any inconsistencies. Therefore, we performed a new experiment using Scenario 2 in which the application source code was compiled with the *gcc* compiler and executed with the same input data used in Scenario 2. The generated graph showed that the curve of the compiled code is equal to the one produced by the tool, which means that there may have been some loss of information when sending data in the hybrid simulation. Additionally, rounding errors cause differences that should be considered because the hybrid simulation and Frama-C analysis were performed on different hardware and with input data that probably also contained small differences.

3) Source Code: The results of the analysis detected thirty unnecessary and/or duplicate library inclusions. Considering that space critical software systems are embedded applications, a good programming practice is to keep the source code as lean as possible. Consequently, keeping the code with unnecessary *includes*, despite not improving the safety assurance, may create difficulties in understanding it, as well as in the documentation and maintenance of the code. The code analysis identified an error related to a global variable that is passed unnecessarily as a parameter to the function. In this case, it is necessary to analyze the variable use and decide between the following options: deleting this function parameter or keeping the variable as a function parameter and define it locally. During the analysis, we identified lines of code without any effect on the results of the application. In this case, removal of the dead code is recommended. Moreover, we identified an incorrect comment that must be revised.

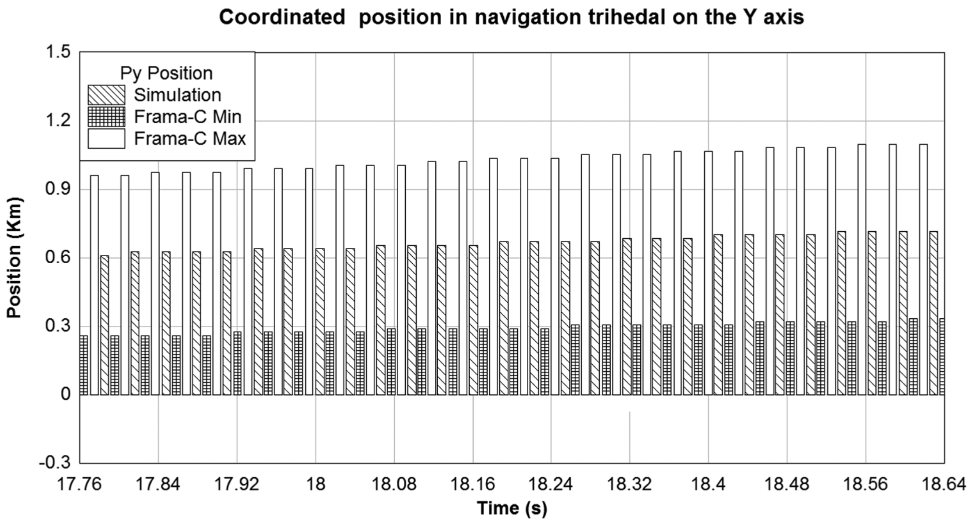


Fig. 16. Bar graph of the position on the Y axis in Scenario 1.

4) Documentation: Indirectly, the code analysis identified discrepancies in the software documentation. In this case, the recommendation is to review the documentation.

5) Performance: Frama-C's value analysis plug-in as a context-sensitive analyzer proved to be suitable for the case study analysis because it was possible to obtain sufficiently accurate results in a short amount of time. Few code annotations were required to support Frama-C's value analysis plug-in to obtain this accuracy; most of these annotations are assertions that were easily discarded.

6) Limitations: In Scenario 1, we encountered difficulties in the analysis of function8f and function8c. The particular combination of value ranges of some variables and certain calculations performed by the underlying algorithms make it extremely difficult to write appropriate annotations that would not be too convoluted to deal with the algorithm logic implemented by this part of the source code. This difficulty does not derive from a tool limitation or an algorithmic error but rather from the inherent difficulty of annotating very complicated code, and it will inevitably occur to some degree in the verification of realistic software.

B. Deductive Verification

1) Source Code: The establishment of a contract via annotations to a RTOS function, invoked by function11, detected the passage of arguments incompatible with the function declaration. Previously, a manual source code inspection also found this error [48]. In this case, correction of the argument type passed to the RTOS function is recommended.

The precondition insertion for a variable detected that function10 passes a parameter in the function11 call, which caused an overflow in this variable. In this case, the developer should be consulted to determine whether there is a reason for this value to be passed.

In function2, the initiation of a floating-point variable requires an implicit typecast. Although there is no implicated consequence for safety assurance, depending on the compiler, an implicit typecast can generate extra code. In this case, it is recommended to initialize variables according to its type to avoid an implicit typecast. Additionally, in the same function, the writing of the precondition identified one redundant *if* conditional statement. Removal of the conditional statement is recommended.

During the analysis, we identified an incorrect comment in function8e. In this case, revising the code comment is recommended.

2) Documentation: The Design Data Dictionary and Software Design Document should be corrected.

3) Performance: In the use of Frama-C's Jessie plug-in, the main activity is the insertion of ACSL annotations in the source code. Despite the amount of annotations for each function being quite significant during implementation of the approach, the use of contracts permits the verification of source code individually in a scalable way.

4) Limitations: Particularly, the case study contains a large number of floating-point computations. Most unproven VCs, approximately 80%, are related to floating-point overflows. One justification is that the floating-point provers are not designed to handle this type of checking automatically. In this case, one solution would be to use a proof assistant, where it is necessary to write the proof manually in an interactive manner. Despite the success in floating-point proofs using a proof assistant [52], in our application context, its usage is not viable at present because it would require significant changes in our work. Nevertheless, we intend to use interactive theorem provers in our future works. In addition, the large number of global variables makes writing the precondition and postcondition more difficult. Regardless, the proposed approach can automatically benefit from any advances and developments made in automatic proof involving floating-point arithmetic.

5) Annotations: Defining annotations to the source code is a difficult task because they must ensure the representation of requirements. Knowledge related to the application domain, the implementation, and annotation language is needed. The Jessie plug-in produces a semantic model of a program through the use of annotations that can be used both in the software documentation and in test case generation. While the insertion of adequate annotations is a valuable source for software documen-

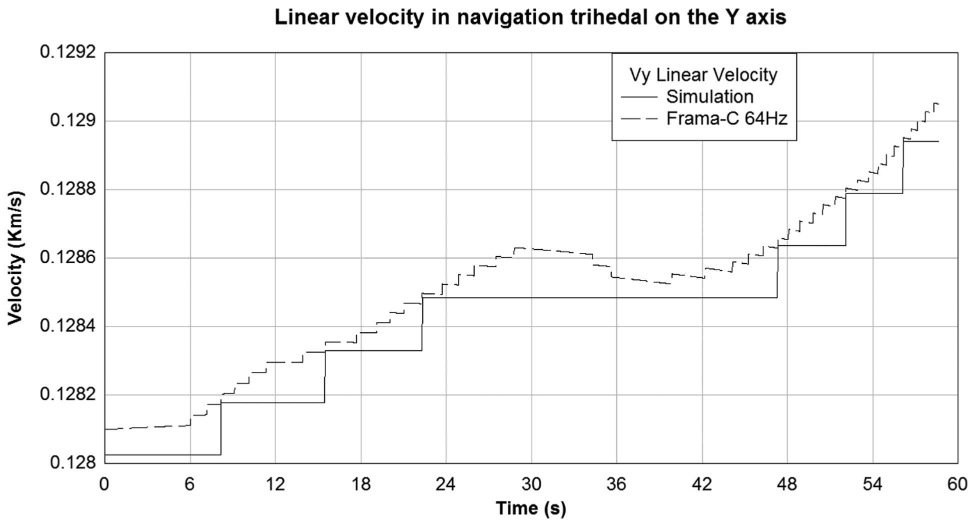


Fig. 17. Comparison graph of velocity in the Y-axis in Scenario 2.

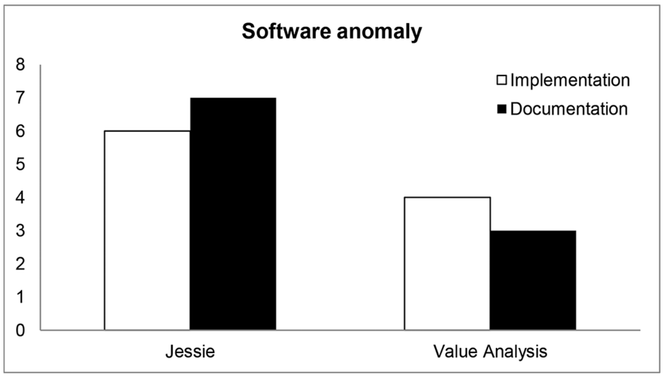


Fig. 18. Impact analysis grouped by area.

tation, we have found that in the present state of development of the tools, in particular of the Jessie plug-in, some annotations have to be inserted that have to do only with technicalities of the plug-in and do not contribute in any useful way to improve the documentation. In addition, the meaning of such annotations is sometimes close to trivial, leaving the user with the feeling that their insertion should not be necessary.

C. Metrics

The metrics derived from the analysis results, which represent the anomalies found in the software, were grouped into two categories: documentation and implementation. Fig. 18 shows the results in this categorization. The bar graph depicts the two impact areas of verification and the number of anomalies detected [53]. We did not detect any errors in the source code with an impact on software safety. This is an expected result considering that the case study is a product of legacy software.

VI. Conclusion

Software quality assurance is essential in the spatial critical systems domain. The challenge is how to successfully achieve this guarantee through software quality attributes such as program correctness because this type of software is generally more complex due to real-time and interface constraints. The typical approach for checking program correctness is to employ the traditional validation activities of testing and simulation. These activities are often not appropriate for finding subtle bugs in the software and cannot generally verify all possible execution scenarios.

We presented an approach, based on abstract interpretation and deductive verification, for formal software verification as an activity to be inserted in a software verification process to complement the validation activities. The use of formal software verification can detect, for example, safety vulnerabilities, which are extremely difficult to test. The executed analysis of the case study detected, in the software documentation and source code, some errors related to comments and descriptions of variable input data and components, which could remain unnoticed through other types of inspection. Although these errors do not have a direct impact on the current result of the application, they can lead to serious future failures. For instance, in a maintenance process, developers could implement some incorrect modifications in the source code following the incorrect documentation.

In summary, the approach may directly contribute to a V&V process and confirms its viability and efficiency when applied to aerospace control software. The execution of a V&V process in critical embedded software systems is mandatory. Any V&V process that is not rigorous, systematic, and methodical may result in incorrect software that can lead to hazardous and/or catastrophic events. Because the proposed approach consists of a set of activities organized in a systematic manner, it can be applied as a tool to help avoid these events, thereby increasing the software reliability. Additionally, the approach requires a deep software understanding both for its execution and for evaluating results, which may help in performing maintenance activities and the software engineering process of new or outsourced software systems of the same type.

Future works include further exploring the use of interactive theorem provers and behavioral specification to assist the testing activity. Furthermore, formal software verification by abstract interpretation could be exploited as a tool for software simulation. Additionally, future work may investigate the use of automated alternative techniques for formal software verification, such as software model checking.

ACKNOWLEDGMENT

The authors would like to thank the Brazilian Space Agency, Institute of Aeronautics and Space, Technological Institute of Aeronautics, HASLab/INESC TEC & University of Minho, MSc. Martha A. D. Abdala, and all the reviewers for their invaluable work on how to improve the paper.

REFERENCES

[1] C. Ebert and C. Jones, "Embedded software: facts, figures, and future," *IEEE Comput.*, vol. 42, no. 4, pp. 42–52, Apr. 2009.

[2] N. G. Leveson, "Role of software in spacecraft accidents," *J. Spacecraft and Rockets*, vol. 41, no. 4, pp. 564–575, 2004.

[3] J.-L. Lions, Ariane 5—Flight 501 Failure, European Space Agency, Paris, France, Tech. Rep., Jul. 1996.

[4] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. S. LaPiana, P. J. Rutledge, D. Folta, and R. Sackheim, Mars Climate Orbiter Mishap Investigation Board—Phase I Report, NASA, Tech. Rep., Nov. 1999 [Online]. Available: ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf

[5] A. Albee, S. Battel, R. Brace, G. Burdick, P. Burr, J. Casani, D. Dipprey, J. Lavell, C. Leisinf, D. MacPherson, W. Menard, R. Rose, R. Sackheim, Al Schallenmuller, and C. Whetsel, Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions—JPL Special Review Board, JPL Jet Propulsion Laboratory, Tech. Rep., Mar. 2000 [Online]. Available: http://spaceflight.nasa.gov/spacenews/releases/2000/mpl/mpl_report_1.pdf

[6] M. Trella, E. L. Herring, H. R. Freeman, W. Kilpatrick, A. Reth, M. Greenfield, J. Credland, R. Laine, D. Machi, and A. Smith, SOHO Mission Interruption Joint NASA/ESA—Investigation Board—Final Report, NASA-ESA, Tech. Rep., Aug. 1998 [Online]. Available: http://sohowww.estec.esa.nl/whatsnew/SOHO_final_report.html

[7] G. C. Buttazzo, *Hard Real-Time Computing Systems Predictable Scheduling Algorithms and Application*, 3rd ed. New York, NY, USA: Springer, 2011.

[8] *Space Engineering—Software*, European Cooperation for Space Standardization Std. ECSS-ST-40C, Mar. 2009.

[9] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate, "Testing or formal verification: DO-178C alternatives and industrial experience," *IEEE Softw.*, vol. 30, no. 3, pp. 50–57, May 2013.

[10] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proc. 36th Annu. ACM/IEEE Design Automation Conf. (ser. DAC'99)*, New York, NY, USA, 1999, pp. 317–320 [Online]. Available: http://doi.acm.org/10.1145/309847.309942

[11] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages (ser. POPL'77)*, New York, NY, USA, 1977, pp. 238–252.

[12] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[13] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects Comput.* vol. 27, no. 3, pp. 573–609, 2015 [Online]. Available: http://dx.doi.org/10.1007/s00165-014-0326-7

[14] L. Correnson, P. Cuoq, F. Kirchner, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski, Frama-C User Manual Release Neon-20140301, CEA List, Saclay, France, 2014 [Online]. Available: http://frama-c.com/download/user-manual-Neon-20140301.pdf

[15] P. Cuoq, B. Yakobowski, and V. Prevosto, Frama-C's Value Analysis Plug-in Neon-20140301, CEA List, Saclay, France, 2015 [Online]. Available: http://frama-c.com/download/value-analysis-Neon-20140301.pdf

[16] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Static analysis and verification of aerospace software by abstract interpretation," *Proc. AIAA Infotech @Aerospace (I@A 2010),* p. 38, Apr. 2010, no. AIAA-2010-3385. American Institute of Aeronautics and Astronautics (AIAA) [Online]. Available: http://www.di.ens.fr/~mine/publi/bertrane-al-aiaa10.pdf

[17] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd your herd of provers," in *Proc. Boogie 2011: 1st Int. Workshop Intermediate Verification Languages*, 2011, pp. 53–64.

[18] P. Baudin, P. Cuoq, J.-C. Filliatre, C. Marche, B. Monate, Y. Moy, and V. Prevosto, ACSL: ANSI/ISO C Specification Language Neon-20140301, CEA LIST and INRIA, FR, release 1.8, 2014 [Online]. Available: http://frama-c.com/download/acsl-implementation-Neon-20140301.pdf

[19] Alt-Ergo., The Alt-Ergo SMT Solver Homepage, 2015 [Online]. Available: http://alt-ergo.lri.fr/

[20] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *J. ACM* vol. 52, no. 3, pp. 365–473, May 2005 [Online]. Available: http://doi.acm.org/10.1145/1066100.1066102

[21] C. Barrett and C. Tinelli, "Cvc3," in *Proc. 19th Int. Conf. Computer Aided Verification (ser. CAV'07)*, Berlin, Heidelberg, Germany: Springer-Verlag, 2007, pp. 298–302 [Online]. Available: http://dl.acm.org/citation.cfm?id=1770351.1770397

[22] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. Theory and Practice of Software, 14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*, Berlin, Heidelberg, Germany: Springer-Verlag, 2008, pp. 337–340 [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[23] Gappa, Gappa (génération automatique de preuves de propriétés arithmétiques) homepage, 2015 [Online]. Available: http://gappa.gforge.inria.fr/

[24] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, "Testing static analyzers with randomly generated programs," in *Proc. 4th Int. Conf. NASA Formal Methods (ser. NFM'12)*, Berlin, Heidelberg, Germany: Springer-Verlag, 2012, pp. 120–125.

[25] Frama-C news and ideas homepage, 2015 [Online]. Available: http://blog.frama-c.com/index.php?pages/Csmith-testing

[26] C. Marche and Y. Moy, "The Jessie plugin for deductive verification in Frama-C," ser. Blue Book, No. 4, INRIA, Toccata, France, Mar. 2014 [Online]. Available: http://krakatoa.lri.fr/jessie.pdf

[27] R. Bonichon and B. Yakobowski, "Frama-C's Metrics Plug-in 20140301 (Neon)," ser. Blue Book, No. 4, CEA LIST. Saclay, France, Jun. 2013 [Online]. Available: http://frama-c.com/download/metrics-manual-Neon-20140301.pdf

[28] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina, "Using bounded model checking for coverage analysis of safety-critical software in an industrial setting," *J. Automat. Reason.* vol. 45, no. 4, pp. 397–414, 2010 [Online]. Available: http://dx.doi.org/10.1007/s10817-010-9172-3

[29] E. Rodriguez, M. Dwyer, J. Hatcliff, and Robby, , G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., "A flexible framework for the estimation of coverage metrics in explicit state software model checking," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2005, vol. 3362, pp. 210–228 [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30569-9_11

[30] D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar, "Generating tests from counterexamples," in *Proc. 26th Int. Conf. Software Engineering (ICSE)*, May 2004, pp. 326–335.

[31] V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, J. Cazin, and G. Durrieu, "Formal verification of critical aerospace software," *J. AerospaceLab*, no. 4, pp. 1–8, May 2012.

[32] V. Prevosto, J. Burghardt, J. Gerlach, K. Hartig, H. Pohl, and K. Voellinger, "Formal specification and automated verification of railway software with frama-C," in *Proc. 11th IEEE Int. Conf. Industrial Informatics (INDIN)*, Jul. 2013, pp. 710–715.

[33] S. Duprat, P. Gaufillet, V. M. Lamiel, and F. Passarello, "Formal verification of SAM state machine implementation," in *Proc. Embedded Real Time Software and Systems (ser. ERTS'10)*, May 2010.

[34] D. Pariente and E. Ledinot, "Formal verification of industrial C code using frama-C: A case study," in *Proc. 1st Int. Conf. Formal Verification of Object-Oriented Software (ser. FoVeOOS'10)*, Jun. 2010.

[35] S. Boldo and T. M. T. Nguyen, "Hardware-independent proofs of numerical programs," in *Proc. 2nd NASA Formal Methods Symp.*, 2010.

[36] C. Hoare, "Viewpoint: Retrospective: An axiomatic basis for computer programming," *Commun. ACM* vol. 52, no. 10, pp. 30–32, Oct. 2009 [Online]. Available: http://doi.acm.org/10.1145/1562764.1562779

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

[38] V. D'silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, 2008.

[39] A. Puccetti, "Static analysis of the XEN kernel using frama-C," *J. Universal Comput. Sci.*, vol. 16, no. 4, pp. 543–553, Feb. 2010.

[40] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin, "Space software validation using abstract interpretation," in *Proc. Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, Istanbul, Turkey: ESA, May 2009, vol. SP-669, pp. 1–7.

[41] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM* vol. 50, no. 5, pp. 752–794, Sep. 2003 [Online]. Available: http://doi.acm.org/10.1145/876638.876643

[42] E. Clarke, D. Kroening, and F. Lerda, , K. Jensen and A. Podelski, Eds., "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2004, vol. 2988, pp. 168–176 [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24730-2_15

[43] T. Ball and S. Rajamani, , G. Berry, H. Comon, and A. Finkel, Eds., "The SLAM toolkit," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2001, vol. 2102, pp. 260–264 [Online]. Available: http://dx.doi.org/10.1007/3-540-44585-4_25

[44] S. F. Siegel and T. K. Zirkel, "TASS: The toolkit for accurate scientific software," *Math. Comput. Sci.*, vol. 5, no. 4, pp. 395–426, 2011.

[45] Instituto de aeronáutica e espaço—projeto VLS-1 homepage, 2015 [Online]. Available: http://www.iae.cta.br/site/page/view/en.vls1.html

[46] W. D. Castro, "Estrutura do sistema de controle do VLS," Controle e Instrumentação, vol. 7, no. 72, pp. 71–77, 2002.

[47] D. S. Carrijo, A. P. Oliva, and W. D. Castro, "Hardware-in-loop simulation development," *Int. J. Model. Simul.*, vol. 22, no. 3, pp. 167–175, 2002.

[48] M. A. Santos, P. Takahashi, and C. H. N. Lahoz, "A process of code inspection for space software," in *Proc. Int. Astronautical Congr.*, Daejeon, Korea, 2009.

[49] Jessie—static variable, 2007 [Online]. Available: http://lists.gforge.inria.fr/pipermail/frama-c-discuss/2013-February/003519.html

[50] Tornado 2.0 Online Manuals—BSP Reference, 1999, ser. Blue Book, No. 4.

[51] 2015, Possible bug in bitwise operators and jessie [Online]. Available: http://lists.gforge.inria.fr/pipermail/frama-c-discuss/2010-February/001794.html

[52] S. Boldo and C. Marche, "Formal verification of numerical programs: From C annotated programs to mechanical proofs," *Mathematics in Computer Science,* vol. 5, no. 4, pp. 377–393, 2011 [Online]. Available: http://dx.doi.org/10.1007/s11786-011-0099-9

[53] *IEEE Standard Classification for Software Anomalies*, IEEE Std. 1044, IEEE Computer Society Std., 2009.

Rovedy Aparecida Busquim e Silva received the doctoral degree in electronics and computing engineering at the Technological Institute of Aeronautics in 2013.

She is a system analyst employed by the Institute of Aeronautics and Space, an organization subordinated to the Department of Science and Aerospace Technology of the Brazilian Air Force, Sao Jose dos Campos, Brazil. Her current research is in real-time systems development and software system development process.

Nanci Naomi Arai received the Master's degree in applied computing at the National Institute for Space Research in 2001.

She is a research assistant at the Institute of Aeronautics and Space, an organization subordinated to the Department of Science and Aerospace Technology of the Brazilian Air Force, Sao Jose dos Campos, Brazil. Her main research interests are software engineering and real-time systems development.

Luciana Akemi Burgareli received the doctoral degree in electrical engineering with an emphasis in digital systems at the Polytechnic School of the University of Sao Paulo in 2009.

She is a system analyst at the Institute of Aeronautics and Space, an organization subordinated to the Department of Science and Aerospace Technology of the Brazilian Air Force, Sao Jose dos Campos, Brazil. Her main research interests are software engineering and real-time systems development.

Jose Maria Parente de Oliveira received the Master's and doctoral degrees in electronics and computer engineering at the Technological Institute of Aeronautics.

He is an Associate Professor at the Technological Institute of Aeronautics. He is a Chief of the Division of Computer Science and Coordinator of the Graduate Program in electrical and computer engineering from Technological Institute of Aeronautics.

Jorge Sousa Pinto received the doctoral degree from L'Ecole Polytechnique, France, in 2001.

He is an Associate Professor at the Department of Informatics of the University of Minho, Portugal, and a senior researcher at HASLab/INESC TEC.