Chapter 14
Exploring Frama-C Resources by Verifying Space Software

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

Abstract The verification process is mandatory in the critical software realm. To improve this process, static analysis tools can make significant contributions. Static analysis meets a variety of goals, including error detection, security analysis, and program verification, which is why standards for critical software development recommend the use of static analysis to identify errors that are difficult to detect at run-time. Thus, this chapter presents a case study on the use of Frama-C as a static analyzer for formal verification of critical software and a lightweight semantic-extractor tool; the former uses an abstract interpretation technique, and the latter allows for the extraction of semantic information to provide a better understanding of source code. In practical terms, the chapter shows how Frama-C can support the development life cycle of an inertial system in aerospace applications, reporting a list of pros and cons. The final results indicate the benefits obtained in terms of software safety, software quality assurance and, consequently, the software verification process.

R. A. Busquim e Silva (✉) · N. N. Arai · L. A. Burgareli
Division of Aerodynamics, Control and Structures (ACE), Institute of Aeronautics and Space (IAE), São José dos Campos, SP 12228-904, Brazil
e-mail: rovedyrabs@fab.mil.br

N. N. Arai
e-mail: nancinna@fab.mil.br

L. A. Burgareli
e-mail: lucianalab1@fab.mil.br

J. M. Parente de Oliveira
Division of Computer Science, Aeronautics Institute of Technology (ITA), São José dos Campos, SP 12228-900, Brazil
e-mail: parente@ita.br

J. Sousa Pinto
High-Assurance Software Laboratory (HASLab), Institute for Systems and Computer Engineering, Technology and Science (INESC TEC) and University of Minho, 4710-57 Braga, Portugal
e-mail: jsp@di.uminho.pt

N. Kosmatov et al. (eds.), *Guide to Software Verification with Frama-C*, Computer Science Foundations and Applied Logic, https://doi.org/10.1007/978-3-031-55608-1_14

Keywords Embedded aerospace software · Formal verification · Frama-C · Software safety · Static analysis

14.1 Is Static Analysis Worthwhile?

A verification process is mandatory in the critical software development life cycle. Static analysis can be included in this process to make it more efficient. Furthermore, it is recommended by space standards and researchers in the scientific community.

The European Cooperation for Space Standardization standard ECSS-E-ST-40C on Space engineering–Software (2009) recognizes that static analysis is a useful resource in terms of code verification activity, e.g., [22]:

> f. 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.

The National Aeronautics and Space Administration Procedural Requirements NPR-7150.2D–NASA Software Engineering Requirements (2022) recommend the use of static analysis to define software management requirements and software engineering life cycle requirements, e.g., [43]:

> 3.11.5 The project manager shall test the software and record test results for the required software cybersecurity mitigation implementations identified from the security vulnerabilities and security weaknesses analysis. [SWE-159]
> Note: Include assessments for security vulnerabilities during Peer Review/Inspections of software requirements and design. Utilize automated security static analysis as well as coding standard static analyses of software code to find potential security vulnerabilities.

> 3.11.7 The project manager shall verify that the software code meets the project's secure coding standard by using the results from static analysis tool(s). [SWE-185]

> 4.4.4 The project manager shall use static analysis tools to analyze the code during the development and testing phases to, at a minimum, detect defects, software security, code coverage, and software complexity. [SWE-135]

According to the International Standard ISO/IEC/IEEE 24765 (2017), static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [37]. From an implementation point of view, static analysis of source code is the science of computing synthetic information about the source code without executing it [4].

Static analysis can be applied to meet a variety of objectives: type checking, style checking, program understanding, program verification and property checking, bug finding, and security review [15].

To meet the aforementioned recommendations related to program understanding, bug finding, and program verification, in this chapter, we exploit the application of

the Framework for Modular Analysis of C (Frama-C).[1] Frama-C gathers several static analysis techniques into a single collaborative framework [4].

The objective of this chapter is to present useful insights into the benefits and challenges associated with the use of Frama-C and its practical feasibility as a tool for use in the software verification process. We aim to share our experience in applying Frama-C in a case study from a real project in the aerospace domain with both the scientific community and software engineering teams interested in verifying their software.

14.2 Literature Review of Frama-C Applications

In the industrial and scientific communities, the most common use of Frama-C is related to plug-ins based on formal methods. An important benefit of formal methods is that they produce sound analyses, i.e., every finding is correct; they never assert a property to be true when it is not true. Formal methods help to find bugs during the early phases of the software development life cycle, particularly through the use of software verification techniques. Formal software verification can be employed in a manner that is complementary to testing and simulation activities to help ensure critical system reliability.

Source code testing and inspection activities can become very expensive depending on the size of the application and the coverage criterion adopted [49]. In addition, techniques such as model checking, run-time verification, deductive verification, and abstract interpretation can be used to complement these activities by trying to maintain the cost-benefit relationship inherent to each of them. Model checking analyzes a program's model automatically; however, it presents the well-known problem of state space explosion [16]. Run-time verification also automatically analyzes a concrete execution of a program but not exhaustively. Deductive verification generates logical formulas (proof obligations) to be proven by other mechanisms in a powerful way, but it is not completely automatic. Abstract interpretation overapproximates the behavior of a program nearly automatically but generates false alarms.

Several works have employed Frama-C to analyze systems from different application areas. Approaches that improve the usability of formal methods in the software development life cycle with Frama-C were presented in [40]. In another related work [20], the main goal was to establish a scalable methodology for using static analysis through the development process by a development team. The authors presented modular analysis and bottom-up strategies involving the Eva and Wp plug-ins. In the safety context, they proposed a framework for safe software development in which the integrated code is analyzed statically to verify the system's critical safety properties using Eva and the formal specification language ACSL [33].

In addition, several published works have demonstrated the application of formal methods in the verification of critical software in specific domains. We select some

[1] https://frama-c.com.

works that apply Frama-C in security- and safety-critical systems in the Internet of Things (IoT), avionics, and nuclear domains.

In the IoT domain, an example of how formal verification can be applied using Frama-C is shown in [7]. The growing number of connected devices around the world, applied or not to critical systems, require security and formal methods as a way to ensure improved reliability. The work is based on a combination of three verification techniques: static analysis to guarantee the absence of run-time errors (Eva plug-in), deductive verification for functional correctness (Wp plug-in), and dynamic verification for parts of the source code that cannot be proven using deductive verification (E-ACSL plug-in).

In the avionics domain, a method of detecting software security vulnerabilities in real time was developed in such a way that it automatically combines static Eva and dynamic E-ACSL analyses in the Frama-C suite [14]. The results confirmed that Frama-C is capable of identifying families of cybersecurity weaknesses and analyzing applications more robustly while delivering the required performance for industrial scale-up. Additionally, formal methods have been applied to the new avionics software products developed at Airbus [9]. The Wp plug-in and the ACSL specification language of Frama-C are used in the verification processes. Finally, the formal verification of the Compact Position Reporting (CPR) algorithm, which is part of the Automatic Dependent Surveillance-Broadcast (ADS-B) system for aircrafts, was performed [21]. The Wp plug-in was used to generate verification conditions, which were discharged with the aid of the Gappa and Alt-Ergo automatic solvers.

In the nuclear domain, a comparative analysis based on the abstract interpretation method of different tools, including the Frama-C Value plug-in, was used to assess safety-critical software employed in nuclear power plants [44]. The article describes practical experimentation and presents an overview of the results and limitations of these tools.

In addition to the aforementioned areas, research projects have applied Frama-C to carry out analyses in industrial case studies. To facilitate the development of safe software systems, an engineering method that consists of four stages was proposed: system modeling and validation, code generation and integration, static code analysis, and dynamic code analysis. In the static analysis stage, the source code was annotated using ACSL, and it was analyzed through the Eva plug-in to verify the critical safety properties. The proposed method was applied in several industrial and research projects [32]. The Verification Engineering of Safety- and security-Critical Dynamic Industrial Applications (VESSEDIA) project proposed to enhance and scale up modern software analysis tools, namely, the mostly open source Frama-C analysis platform, to allow developers to rapidly benefit from them when developing connected applications and to provide safety and security to many new software applications and devices [55].

Some works have evaluated Frama-C by comparison with other static analysis tools. Frama-C was appointed as one of the best open-source static analysis tools available for C even though it is not specifically focused on security [29]. Other research has evaluated how much time tools, including Frama-C, need to detect

run-time errors in programs with and without slicing. In the case of Frama-C, the Slicing plug-in improved the verification process time [39].

Frama-C has also been used as a basis for developing the commercial tool TrustInSoft Analyzer, which is a C and C++ source code analyzer and the industrial version of the open-source Frama-C [54].

Two important aspects of Frama-C that must be considered are (1) its capability to satisfy tool qualification requirements, which shows that it is a reliable tool and worth the effort to use, and (2) its ability to help applications achieve the requirements established in certification standards, including, for example, those established by organizations such as the National Institute of Standards and Technology (NIST) and Bureau Veritas.

The Frama-C Eva plug-in is one of the few tools in the world to have passed the NIST 6th Static Analysis Tool Exposition (SATE VI) Ockham Sound Analysis Criteria. According to the three evaluated criteria, Frama-C Eva: (1) is a sound tool; (2) produces at least 75% of the program findings; and (3) all findings are correct [6]. A partnership between Bureau Veritas and CEA-List produced a guideline for Software Development & Assessment using Frama-C as part of its proof of concept and concluded that Frama-C makes it possible to analyze and verify software to ensure it meets recommended standards at optimal cost [12]. Furthermore, Frama-C is able to correlate outputs with the entries of the Common Weakness Enumeration (CWE) and of the SEI CERT C Coding Standard [17].

The fact that the Frama-C Eva plug-in has also satisfied the SATE VI criterion is an indicator that it can assist in accomplishing the certification requirements of critical software. A source code analysis activity must achieve two certification objectives: (1) accuracy and consistency and (2) verifiability [47]. The former determines the correctness of the source code (e.g., floating-point arithmetic, use of uninitialized variables, and unused variables), which can be shown by the absence of some classes of run-time errors that were established through the technique of abstract interpretation. The latter ensures that the source code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it.

14.3 Overview of IAE Satellite Launcher Projects

Brazilian space projects aim at scientific technological research, innovation, and development to consolidate national air and space power, space launch operations, and services in aviation, space and defense systems [41].

The Institute of Aeronautics and Space (IAE) is a military organization of the Aeronautical Command (COMAER) within the Department of Aerospace Science and Technology (DCTA). As shown in Fig. 14.1, IAE and several public and private institutions are involved in the space program coordinated by the Brazilian Space Agency (AEB) [10]. IAE has developed two satellite launcher projects and a series of sounding vehicles (VS) [1].

Fig. 14.1 Institutions coordinated by AEB [10]

IAE has been developing satellite launcher projects, as illustrated in Fig. 14.2. The Satellite Launch Vehicle 1 (VLS-1) was the first Brazilian satellite launch vehicle. This project aimed to provide Brazil with the ability to design, manufacture, launch, control, stabilize, and deliver a payload in orbit autonomously. A real-time embedded software, named SOAB, was developed for VLS-1 to control flight based on a set of control, navigation, and guidance algorithms as well as the sequence of events that occur in the various launching phases. SOAB was also responsible for sending telemetry data to the ground station [36].

Due to technological challenges and limited financial and human resources, the need to realign the VLS-1 development strategy has arisen, taking advantage of its legacy and adapting it to the current scenario. The new proposal encompasses the development of a simpler controlled suborbital vehicle [36].

The Microsatellite Launch Vehicle (VLM-1) project aims to develop a three-stage solid propellant vehicle for the launch of microsatellites in equatorial or re-entry low Earth orbits (LEO). IAE and the German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt–DLR) are jointly developing the VLM-1. The Brazilian part of the project first comprises the development of VS-50 for in-flight qualification of the S50 motor, the electrical and pyrotechnic networks, and the guidance, control and navigation system that will later be used in VLM-1, mitigating the technical risks of the project [1, 2]. The VS-50 project concerns a two-stage solid propellant suborbital vehicle. The first stage of both vehicles is almost identical, and the second stage of VS-50 has the same configuration as the third stage of VLM-1 [1].

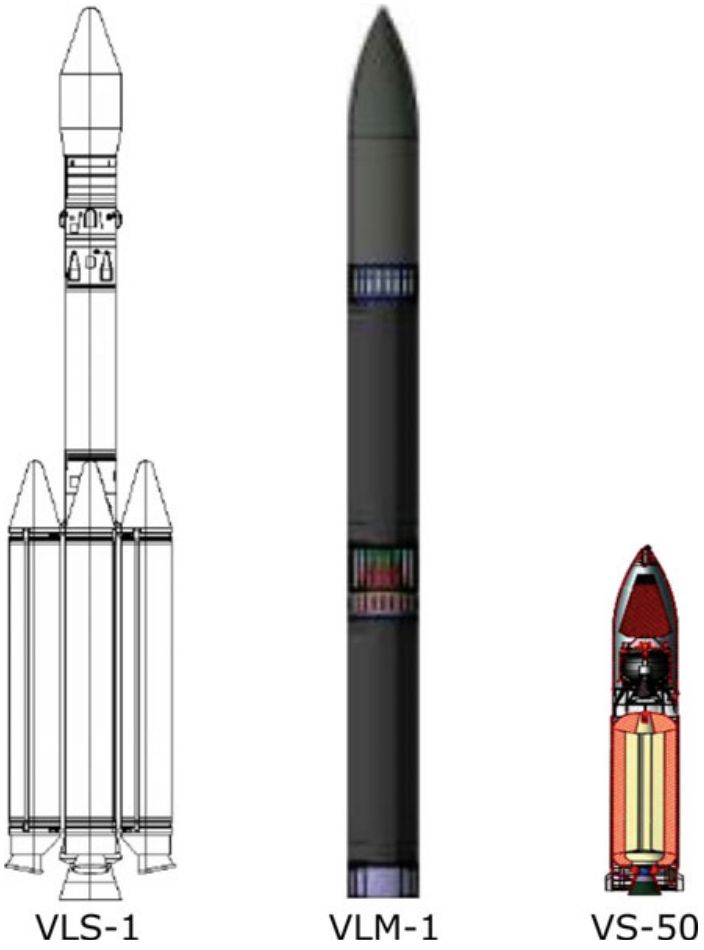

Fig. 14.2 IAE Launch vehicle projects. Adapted from [1]

14.3.1 Case Study: Embedded Aerospace INS Software

During the development of launch and suborbital vehicles at IAE, the need for reliable inertial navigation, guidance, and control systems has arisen [35]. Therefore, the SISNAV (*Sistema de Navegação Inercial*–inertial navigation system) is under development to provide a national inertial navigation system (INS), replacing the current imported system. To avoid a trade embargo, the objective is to update the involved technology. Initially, conceived to be used in VLS-1, SISNAV will be a subsystem in VS-50 and VLM-1 [35, 45].

An INS consists of an inertial measurement unit (IMU) and a computational unit. The SISNAV is a strapdown INS whose IMU has three accelerometers and three fiber optic gyrometers (FOGs) that measure acceleration and angular velocity, respectively. SISNAV's computational unit encompasses a floating-point digital signal processor (DSP) with 256K internal memory, a field-programmable gate array (FPGA), a voltage-to-frequency (V/F) card (accelerometer interface), serial channels (connecting to FOG), and additional A/D and I/O interfaces [45].

SISNAV's software (SSISNAV) is a critical real-time software embedded in a DSP that processes data from the IMU and performs calculations to determine position,

velocity, and attitude. SSISNAV implements algorithms for testing, calibration, and compensation of inertial sensor signals, self-alignment, navigation, and data storage and transmission [52]. A Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) code embedded in FPGA is responsible for data acquisition, which is out of the scope of this work.

SSISNAV is a natural candidate for Frama-C analysis because it is a highly critical embedded software implemented in the C programming language. Embedded code is adequate to be analyzed by Frama-C due to its characteristics, such as the absence of recursion, dynamic memory allocation, and calls to external libraries. SSISNAV source code consists of ninety-nine functions and 43,699 physical source lines of code (SLOC). It contains main algorithms (accelerometer and gyrometer calibration, coarse and fine alignments, and navigation) with many numerical calculations based on floating-point arithmetic, whose computation depends on six C library mathematical functions. The implementation contains function-like macros and static memory allocations. The SSISNAV team has been testing and modifying the source code and extending the documentation. Its development follows the processes proposed by the ECSS standards [22, 24].

Components have been developed and tested individually in the Laboratory of Identification, Control and Simulation (LINCS) [35, 45]. Furthermore, integrated configurations have been tested in roller coaster rides, aircraft flights and compact electric utility vehicle rides. The plan is that SISNAV will be embedded in a Brazilian launcher in a future flight. A series of tests is planned in advance, including a flight as a payload onboard the vehicle HANBIT-TLV, which was developed by the South Korean company INNOSPACE [34]. The performance of SISNAV is assessed using a hybrid simulation environment. This is a hardware-in-the-loop simulation of the complete system, including the following components: the onboard computer with the control laws, the vehicle dynamics, the actuator dynamics, and the sensor dynamics [13]. Since the simulation data are recorded according to the state, preparation or navigation, the static code analysis is executed in two parts. We used the simulation output obtained by the control system team. Because the data of the selected case study are classified, this work presents only a limited subset of the results. For the same reason, some information pertaining to the algorithms (such as function names) is not disclosed.

14.4 Employing Frama-C in Space Software

This section presents our experiments in applying Frama-C to space software. First, we provide an overview of the results obtained from our former experiment, whose approach relied on abstract interpretation and deductive verification techniques. Second, we focus on presenting our current experience in using Eva as a static analyzer for formal verification and Callgraph and SpareCode as lightweight semantic-extractor tools. Our current approach to static analysis by abstract interpretation slightly improves on our previous work, and we present here in detail our experi-

ments in the context of SSISNAV. Furthermore, this section describes the tools and equipment used, the experiments carried out, and the results obtained.

14.4.1 Former Experiment: Embedded Aerospace Control Software

Our first experience in applying Frama-C started with a research project whose objective was to develop an approach to formal verification activity in the context of spatial critical software. Our case study was the SOAB, which is the software onboard the VLS-1. The results suggested that this activity is relevant in the software verification process and can be used to complement validation activities, including testing and simulation [50].

The main contribution of this research project was to disseminate our experience in applying formal software verification to the scientific community and to software engineering teams interested in verifying their software. We used formal static analysis based on two techniques: abstract interpretation [18] and deductive verification [31], which were implemented using Frama-C through the Value (currently named Eva) and Jessie plug-ins, respectively. Moreover, InOut and Metrics were used as auxiliary plug-ins.

At the end of the research project, the Frama-C analysis did not emit any alarms. This result was expected considering that the case study concerned a product that had been in production for many years. Nevertheless, we detected anomalies related to two software products: documentation and implementation. According to Value, three anomalies in the documentation and four in the source code were detected, while Jessie allowed us to detect seven anomalies in the documentation and six in the source code. Although the results obtained with Frama-C are limited to alarms, they can indirectly lead to identifying anomalies in the software documentation and implementation. Both kinds of anomalies were detected in our approach using Value and Jessie.

Our experience with the code inspection process for space software made it relatively easy and intuitive to detect anomalies visually (specifically implementation anomalies). The anomalies detected in the source code were related to unnecessary and/or duplicate library inclusions, dead code, redundant code, global variables passed as function arguments, inconsistency in the argument type between the function definition and function body, implicit typecasts, and incorrect comments.

Anomalies in the documentation were discovered by using Value when information about the sensor input data was needed. In the data dictionary (DD) of the software requirements specification (SRS), an incorrect variation domain was discovered, and it was related to sensor input data that were examined to parameterize the analysis. In the software design document (SDD) and DD of the SDD, the descriptions of two sensor vector components were inverted, and anomalies in documentation were found when ACSL function contracts were required to use Jessie.

In the SDD, anomalies in the structure charts were identified: one missing output variable and two incorrect parameters. In the DD of the SDD, there were anomalies in the type definition and names of variable and function parameters.

Perhaps these anomalies could remain unnoticed by other approaches to inspection. Although they did not directly impact the final result of the application, they might create difficulties in understanding, documenting, and maintaining the source code, which could potentially lead to serious failures in the future. For instance, during the software maintenance process, developers might implement some incorrect modifications in the source code based on incorrect documentation. Additionally, finding and reporting anomalies can assist in the evaluation of quality attributes such as reliability and productivity and in process improvement.

The preliminary results were obtained by using the Frama-C 6 (Carbon) release. The final results were later published in [50] with results updated using the Frama-C 10 (Neon) release.

14.4.2 Current Experiment: Embedded Aerospace INS Software

The context of the current selected case study is different from the previous one; SOAB was a legacy project, while SSISNAV is currently under development and has not yet been embedded in a launcher and evaluated during flight. On the other hand, we were able to reduce the effort necessary to carry out the current verification because it is based on the approach that was previously developed for the formal verification activity of SOAB, which provided us with a background for the abstract interpretation technique and familiarity with the tool. In addition to Eva and Metrics, which were already used in the previous project, the use of Callgraph and SpareCode is also investigated in this new case study. Frama-C plug-ins can be categorized according to the intended analyses. Specifically, the plug-ins employed in our experiment can be divided into the following [5]:

- Verification plug-in–Eva;
- Understanding plug-ins–Callgraph and Metrics;
- Simplification plug-in–SpareCode.

Frama-C is a collaborative platform, and some plug-ins work on results that were already computed by other plug-ins in the framework [17]. This dependency relationship can help the user establish a sequence for running the plug-ins. SpareCode relies on Eva except for the `-sparecode-rm-unused-globals` option. Metrics relies on Eva for the `-metrics-eva-cover` and `-eva` options. Callgraph does not depend on Eva, but its precision may be improved by using the `-eva` option. Figure 14.3 shows which plug-ins (identified by their prefix) are running according to the command and options entered by the user in the terminal. The figure also shows the dependencies (or absence thereof) and running order among the employed plug-ins. Parsing of files is executed by the kernel independently of any plug-in.

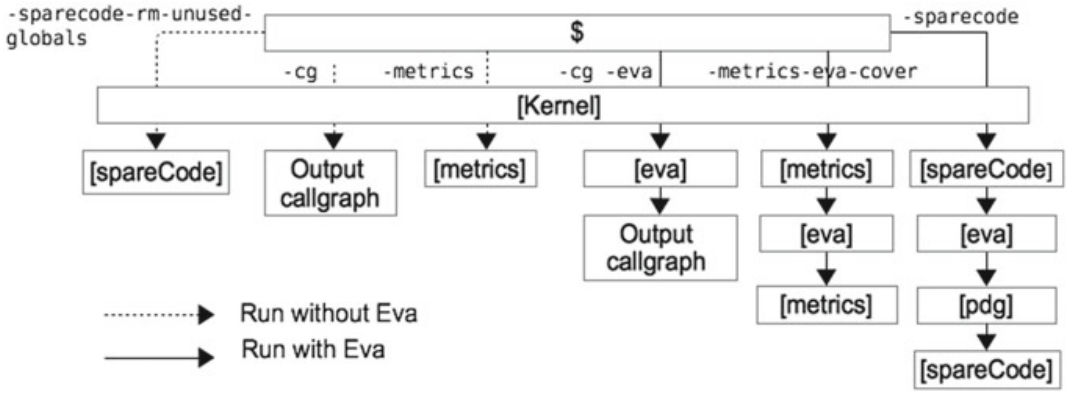

Fig. 14.3 Examples of dependencies among plug-ins running on the terminal

To conduct the SSISNAV experiments, we used the Frama-C 23 (Vanadium) release on a 2.9 GHz notebook with 8 GB of memory and a 1 TB hard disk running the MacOS X Yosemite 10.10.5 release. To plot the graphs, we used Plot2 version 2.6.17. To translate from DOT to PNG format, we used dot/graphviz version 2.40.1. To draw figures, we used the Papyrus release 2022–03 and Apache OpenOffice 4.1.2.

14.4.3 SSISNAV Analysis with the Callgraph Plug-In

The call graph is a directed graph that represents relationships between functions in the program: the nodes are functions, and the edges represent one or more invocations of a function by another function [48]. Call graphs can be used to aid software engineering teams in the following tasks: software documentation to improve program understanding [25], detection of program execution anomalies [28], and analysis of the impact of program changes [42].

The Callgraph plug-in automatically computes the program's call graph [3]. Its output is saved to a dot file [27] that requires an appropriate application to be visualized, or it can be translated into the PNG format.

We have generated call graphs for SSISNAV main functions with some auxiliary functions omitted for simplification purposes. For example, Fig. 14.4 shows the call graph of the main navigation function, which can be generated and translated into the PNG format with the following command lines:

```
$ frama-c -cg out.dot./DIR1/f1.c./DIR2/f2.c./DIR2/f2.h
$ dot out.dot -Tpng -o f1.png
```

Figure 14.5 shows the call graph of the fine alignment function, and it can be observed that the *function5b4* is not called. Therefore, it is necessary to look at other call graphs to investigate if it is called in any other source code file; otherwise, it should be reported as an anomaly. A complete call graph generated from a main function could be used to detect functions that are never called. However, this call graph could be very large and difficult to visualize. Therefore, we have generated

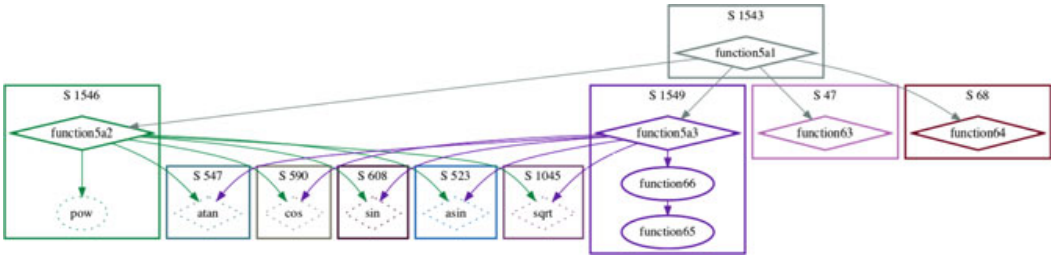

Fig. 14.4 Call graph of the main navigation function

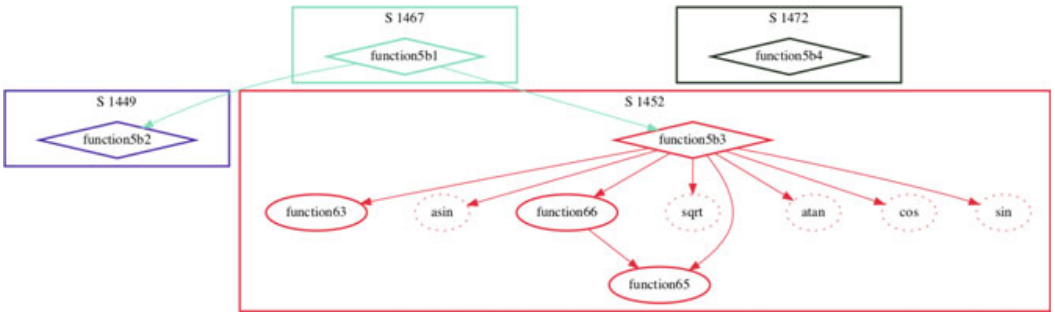

Fig. 14.5 Call graph of the fine alignment function

call graphs for the functions encoding the main algorithms, which are smaller and easier to examine.

After analyzing the applicability of call graphs in our context, we identified that call graphs can assist in writing the SDD section <*5.4.6*><*Subordinates*>, help detect functions that are never called, and contribute to program understanding because they depict the structure of the program with all subordinate functions.

The plug-in does not handle function-like macros, i.e., they were not shown in the call graphs. Additionally, we noticed that a missing function definition is shown as a dotted line, and any function called from it is not shown in the graph, generating an incomplete call hierarchy. This depends on how the source code files are organized. If all function definitions are implemented in a single file, the Frama-C command line will require only this file to generate a complete graph; otherwise, all of the files will be required. The #include directives of the analyzed function, e.g., for navigation or fine alignment, can be examined to select the necessary files.

The option `-eva` generates a call graph according to the analysis, so some branches may not be reached and functions called from them will not be shown. Comparing different call graphs generated with and without the `-eva` option for valid and invalid intervals can be useful in assessing the coverage of functions and visually identifying unused source code.

14.4.4 SSISNAV Analysis with the SpareCode Plug-In

The SpareCode plug-in removes *spare code*, i.e., code that does not contribute to the final results of the program, and it produces an output program that is guaranteed to be a compilable C program [4].

Attention

A piece of source code that does not contribute to the final state of the program is not necessarily dead code; we can only be sure that all its side effects are shadowed by further instructions [53].

The plug-in includes the option `-sparecode-rmunused-globals`, which removes unused global types and variables. Nevertheless, as it prevents the global variables from being listed explicitly, the generated code requires inspection (visually or through a file comparison tool) to detect unused variables. According to a discussion in the Frama-C forum, the debug key `globs` must be enabled, and the debugging level must be assigned the minimum value of 2 to generate the output, including unused global variables preceded by the expression `remove var`. Thus, the `grep` command can be employed to extract only the list of unused variables.

Detecting local static variables is slightly different. According to the same forum discussion, unused local static variables are removed while parsing the source code regardless of whether SpareCode is executed or not. Nevertheless, it is possible to extract the desired information using the option `-kernel-msg-key parser:rmtmps`.

There are two reasons to detect unused variables. First, this is one of several items that may be helpful as preliminary steps to obtain guarantees of code correctness [47]. Second, for embedded applications, a good programming practice is to keep the source code as lean as possible. Despite not improving safety assurance, it might impact the consumption/optimization of memory and impair code readability.

The command used to detect declared but unused global variables in the navigation algorithm in SSISNAV, and its output are the following:

```
$ frama-c navigation.c -sparecode-rmunused-globals -sparecode-msgkey globs -sparecode-debug 2 | grep "remove var"
[sparecode:globs] remove var senLc_2
[sparecode:globs] remove var senLc_4
```

For the navigation algorithm, there were two declared but unused global variables detected in the source code; for the sensor calibration and alignment algorithms, the plug-in did not detect any declared but unused global variables.

To detect declared but unused local static variables, the command and its output for the gyrometer calibration algorithm are given as follows:

```
$  frama-c -main function3a1 calibrationG.c -sparecode-
rmunused-globals -sparecode-msgkey globs -sparecode-
debug 2 -kernel-msg-key parser:rmtmps | grep "removing
local"
[kernel:parser:rmtmps] removing local: proc_giros_dado-
Giros
```

For the coarse alignment algorithm, the command and its output are the following:

```
$    frama-c -main function5c1 alignment1.c -sparecode-
rmunused-globals -sparecode-msgkey globs -sparecode-
debug 2 -kernel-msg-key parser:rmtmps | grep "removing
local"
[kernel:parser:rmtmps] removing local: alingr_ciclo_M2_
dVc
[kernel:parser:rmtmps] removing local: alingr_ciclo_M2_
dThetac
```

The number of declared but unused static local variables detected for the gyrometer calibration and coarse alignment algorithms was one and two, respectively; for the accelerometer calibration, navigation, and fine alignment algorithms, the command execution did not detect any unused local static variables.

14.4.5 SSISNAV Analysis with the **Metrics** *Plug-In*

Metrics are the only way to quantitatively assess the quality of development processes and products, and they are typically used to manage development [23, 24]. It is recommended to generate at least basic metrics such as size (code), complexity (code) and test coverage. An important metric is the cyclomatic complexity, which can directly affect the quality and maintenance of a source code. Standards recommend that safety-critical software components have a defined value of cyclomatic complexity [23, 43].

The Metrics plug-in implements the automatic computation of a set of measures on the source code, e.g., McCabe's cyclomatic complexity, Halstead complexity, SLOC, and the Eva coverage estimate [4]. The option `-metrics-eva-cover` provides the Eva coverage statistics, which include the semantic and syntactic reachability coverages, the coverage estimation, and the number of statements reached in each analyzed function. It can be used to compare the code effectively analyzed by Eva with what Metrics are considered reachable from the main function [8]. Therefore, the user is able to perform a coverage analysis to count how many unreached functions exist, accounting for the number of syntactically reachable functions. Additionally, Metrics provides the number of lines of code containing calls to non-analyzed functions.

The SSISNAV metrics shown in Table 14.1 are delivered to the software quality team because these metrics allow them to perform risk management, indicating whether risks have to be minimized and controlled and thereby reported to developers.

Table 14.1 Global metrics by phase in Scenario 2

	Accelerometer calibration	Gyrometer calibration	Coarse alignment	Fine alignment	Navigation
Cyclomatic complexity	4	6	30	28	25
SLOC	34	67	124	166	148

In our software development processes, there is no requirement related to complexity based on the cyclomatic complexity metric. Therefore, we adopted a maximum cyclomatic complexity level of 20 based on standards [23, 43]. It is necessary to assess the source code when this threshold is exceeded to determine whether it is acceptable or needs to be modified. For the latter, the suggested corrective actions, in addition to reviewing the algorithm logic to reduce complexity, include writing smaller functions, minimizing the number of decision structures, and removing unused source code.

14.4.6 SSISNAV Analysis with the Eva Plug-In

In the Frama-C 15 (Phosphorus) release, the support for the legacy Value plug-in was abandoned and replaced by Evolved Value Analysis (Eva), which provides more precise and extensible abstract domains [11]. The Eva plug-in is a forward dataflow analysis based on the principles of abstract interpretation, and it performs whole-program analyses [38]. Eva is both context-sensitive and path-sensitive [11], i.e., function calls are handled through a symbolic inlining of function bodies [38] and infeasible paths are excluded. Its main objective is to automatically compute sets of possible values for the variables in an analyzed program [11]. An additional purpose of the plug-in is to find alarms, i.e., errors that could occur at run-time and/or to demonstrate their absence. It might be employed to assist in the following tasks [11]: familiarization with foreign code, automatic document production, bug detection, and guaranteeing the absence of bugs.

With respect to our previous research project, the workflow of the software verification approach employing Eva was updated in relation to its inputs and outputs, which are highlighted in red in Fig. 14.6.

The approach begins with preparing the application context, i.e., identifying the information required to perform the verification. There are two scenarios in which the sensor measurements are treated differently. Scenario 1 considers the range of maximum and minimum values accepted by the sensors, which illustrates a typical application of Eva. Scenario 2 is an unusual application of Eva in which it is used as a C interpreter [19]. It becomes very useful when a specific input dataset is known in advance [26], e.g., the values of sensor measurements obtained from a simulation. These values are deterministic, i.e., a specific value obtained from the sensor is

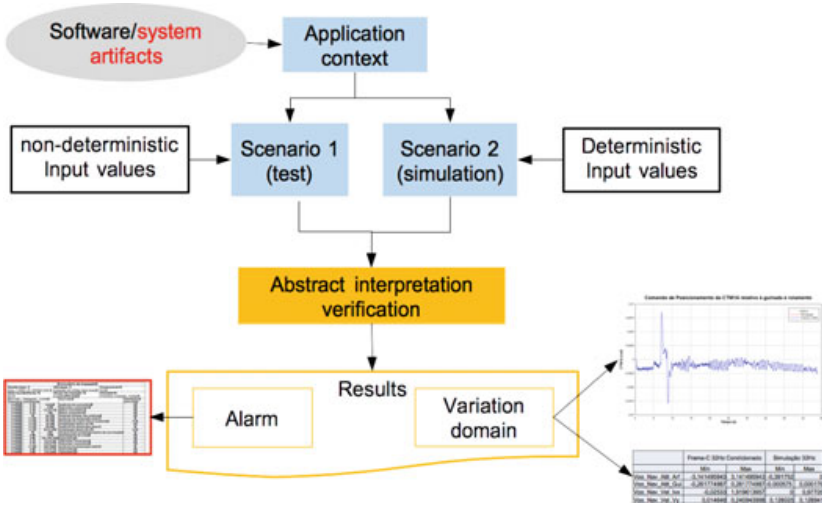

Fig. 14.6 Updated approach overview for verification by abstract interpretation with the Frama-C Eva plug-in. Adapted from [50]

considered for each instant, and thus, Frama-C acts as a simulation tool. Scenario 2 can adequately run a sufficiently deep analysis; it is completely deterministic and does not generate false alarms, while undefined behaviors (not detectable by testing) are still detected. The output results include alarms and the variation domains for the variables, which are presented in graphs and tables to facilitate data visualization and analysis.

The detailed approach is presented as an activity diagram with the updates highlighted in red in Fig. 14.7. The activities are grouped into two main phases: *context definition* and *implementation and refinement*. The former consists of executing the following activities: defining the entry point function, identifying the input data, and identifying the output data to be evaluated. For each inserted function, the latter consists of addressing library function inclusion, addressing missing functions, addressing alarm/nonalarm/message, and analyzing the variation domain. This process is iterative and interactive because it requires user intervention, and it is performed repeatedly until the intended result has been obtained. The user could adjust the accuracy of the analysis through the plug-in parameters to account for the trade-off between efficiency and accuracy.

In the context definition phase, an artifact set that may vary from project to project due to each particular need and availability is required. Some artifacts that assist in this phase are described below:

- Data flow diagram (DFD) from SRS, which contains the input data needed to execute the algorithms and their output data.
- DD from SRS, which contains detailed descriptions of each control and data flow.
- SDD, whose software component design section identifies the variables in the source code.
- DD from SDD, which contains the variable ranges (i.e., maximum and minimum values).

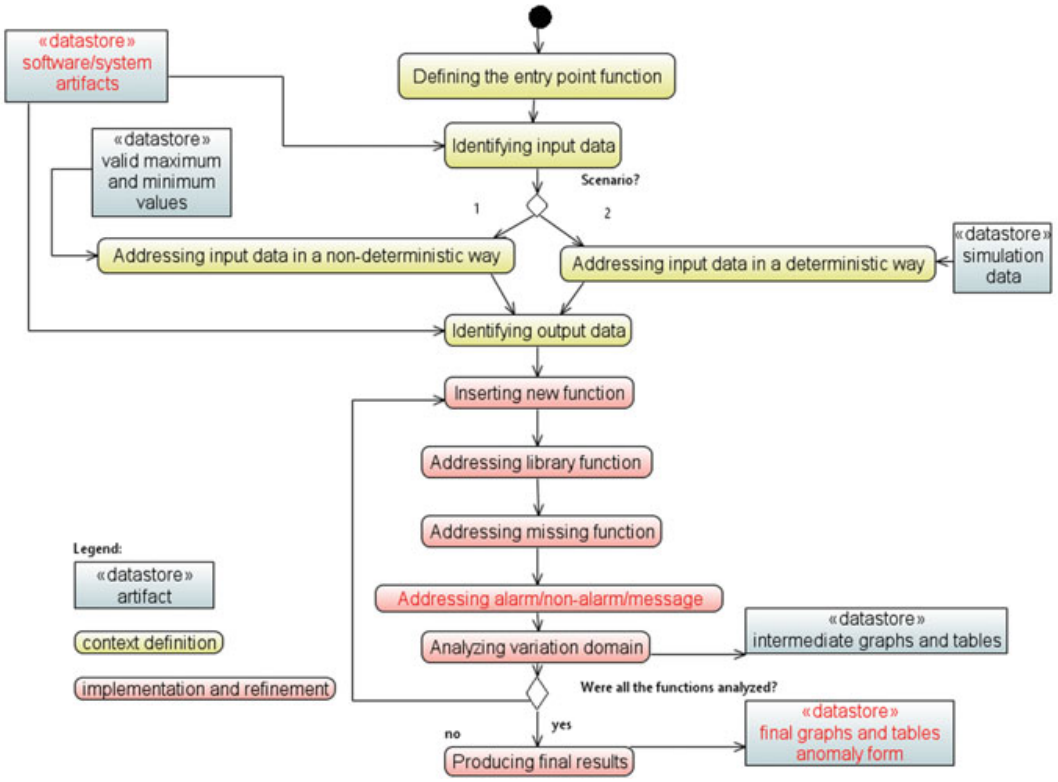

Fig. 14.7 Updated activity diagram for verification by abstract interpretation with the Frama-C Eva plug-in. Adapted from [50]

- System documentation, which can be useful for understanding source code that implements specific algorithms.
- Source code whose implementation itself and/or comments can assist in the identification of data, algorithms, etc.

Thus, with respect to our previous project, the artifact set was updated from *software documentation* to *software/system artifacts* to include system documentation and source code.

In the implementation and refinement phase, we expanded the activities from *Addressing alarms* to *Addressing alarm/non-alarm/message*, given that we are also handling nonalarms and messages that may correspond to loss of precision. An alarm is a guard against some undesirable behavior and corresponds to a particular warning category [3]. In practice, alarms are warning messages emitted by Eva that start with the prefix `[eva:alarm]`. Additionally, there are warning messages that are classified as nonalarms, starting with the word *Warning*. Finally, a message is only informative text that does not begin with the word *Warning* and is not prefixed with `[eva:alarm]` [11]. In relation to the output update, we have added an anomaly form that gathers all of the emitted alarms, which is described in Sect. 4.7.

14.4.6.1 SSISNAV Formal Verification

Eva analysis is executed in two parts corresponding to the preparation and navigation states. The sensor calibration and alignment algorithms are verified during the preparation state, and the navigation algorithm is verified in the navigation state. The experiments were run through the Eva plug-in batch mode.

Following the activity diagram shown in Fig. 14.7, we first describe the context definition phase. Figure 14.8 depicts the required input artifacts to each activity step to define the context for SSISNAV compared to SOAB. The software artifacts employed are SRS, SDD, and the source code, while the system artifact concerns the documentation related to the implementation of SSISNAV algorithms. The activities in this phase are succinctly described in the following paragraphs.

For the *defining the entry point function* activity, the main function in the source code was identified as the entry point function, which was corroborated by its call graph.

In the *identifying input data* activity, we used the context diagram (level 0 DFD)–SRS as an initial source together with the DD–SRS to understand the flows, as shown in Fig. 14.8. The input data required are mission, configuration, and sensor data. For both states and scenarios, it is necessary to set parameters related to specific mission requirement data, e.g., sensor calibration; reference data such as Earth's rotation; and launch site data such as local latitude. We used acceleration, angular velocity, and temperature (all raw data) as sensor input data in the preparation state and the calibrated sensor measurements in the navigation state. After that, we mapped the data/control flow into variables with the aid of the SDD, source code, and system documentation.

To *address the input data in a nondeterministic way*, we asked the system analyst to identify the numeric domain (range) of the input variables. We modified sensor reading functions, replacing hardware function calls with Frama-C built-in primitives to address intervals.

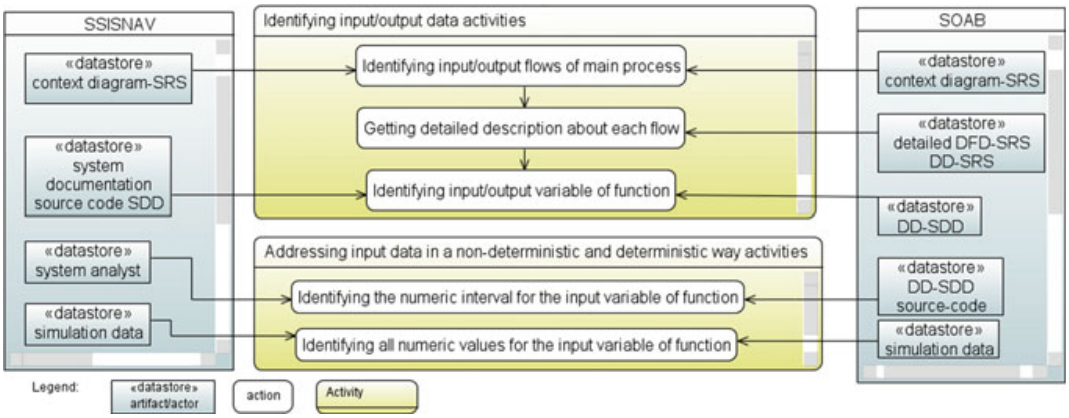

Fig. 14.8 Diagram for context definition of SSISNAV compared to SOAB

To *address the input data in a deterministic way*, we extracted deterministic values from two CSV simulation files, which contain 25,828 and 7,644 rows, for the preparation and navigation states, respectively. We wrote an additional program to extract data from these CSV files and created C functions that assign the extracted data to arrays, emulating the hardware functions. Then, the sensor reading functions were adapted to include these C functions in place of the hardware function calls.

When *identifying output data*, we used the context diagram–SRS and the DD–SRS. We used the following sensor output data: calibrated sensor measurements in the preparation state and inertial linear velocity, inertial position, and Euler angles in the navigation state. We modified the function that composes the telemetry data frames, replacing hardware function calls with Frama-C built-in primitives to display the telemetry data frames.

Now, we discuss the implementation and refinement phase and describe the related activities.

In the *inserting new function* activity, as proposed in our approach, we insert functions one by one to perform the analysis incrementally. In this activity, the *loop* global metric (i.e., number of loops) can be used as an input indicator of the number of loop unroll annotations required for each function.

During the *addressing library function* activity, we performed library inclusion treatment because Eva does not support certain external libraries (e.g., the hardware library). Therefore, because keeping these libraries in the code implies the introduction of a dependency chain that could not be provided, we decided to remove all included directives from the source code and later reinsert them incrementally as needed.

For the *addressing missing function* activity, we identified two missing function types. The input hardware functions were treated in one of two ways: either parameterizing the analysis by providing value intervals through Frama-C functions or using simulation data. The output hardware function was replaced by Frama-C functions to allow the results to be observed. We noticed the introduction of recent Frama-C improvements in its math library. Currently, Frama-C provides implementations for the six employed math functions that are different from our former experience when we had to write C code for functions such as sin, fabs, and modf.

In the *addressing alarm/non-alarm/message* activity, we evaluated and addressed the alarms as soon as they were detected, and we subsequently reran the analysis. Eva detected alarms related to accessing out-of-bounds indices in the navigation init function, uninitialized variables in the coarse alignment function, and NaN or infinite floating-point values in the fine alignment function. These results are presented in Table 14.4, identified by the expression Eva alarm. In addition to those alarms, others related to infinite floating-point values were detected in the alignment and navigation functions, which could not be eliminated. Nevertheless, Eva terminated the analysis, resulting in some intervals corresponding to the data type ranges that cannot be considered sufficiently accurate when evaluating the domain of an output variable.

For example, the emitted messages *starting to merge loop iterations* indicate the necessity of unrolling a loop. Instead of the option `-eva-slevel`, which was used in the previous experiment, we decided to annotate the loops because

the source code is under development and it was easier for us to edit it and test other ways to unroll loops. Adding loop unroll annotations is more precise and stable than `-eva-slevel`; however, it has the drawback of requiring source code changes [11]. Even if modifications to the source code preserve the semantics, it is still recommended to keep the code used for testing and verification as close as possible to the original. For the preparation and navigation states, the numbers of loop unroll annotations inserted were 34 and 33, respectively. The nonalarms emitted by Eva are related to implicit function declarations, missing function specifications, and floating-point constant representations.

Comparing our previous and current Frama-C experiments, we noticed that the obtained nonalarms and messages are quite different for reasons related to differences in the case studies (e.g., implementation particularities) and to the evolution of Frama-C itself. This can make it difficult to use the *grep* command to obtain a specific message from a log file if the displayed messages are unknown. For this reason, it is necessary, after every analysis execution, to visually look for messages in the log file and catalog them. Thus, familiarity with Eva and its previous application in similar case studies make this activity easier.

For the *analysis of variation domain* activity, we applied the following treatment to make the analysis more precise: we used the option `-eva-slevel` to improve the analysis precision when evaluating *if* or *switch* conditional statements. In its default configuration, Eva produces results that are too approximate to handle loops. Loop unroll annotations ensure that Eva unrolls the loops and keeps the analysis precise; otherwise, Eva might generate alarms due to imprecisions [11]. After this treatment, the variation domain for the variables needs to be evaluated, and the analysis continues until valid values for the observed magnitudes have been obtained. To help in this evaluation, we observed intermediate graphs of some variables. We wrote an auxiliary program to extract data from the Eva output to be plotted in graph form.

In the *producing final results* activity, the final results of the verification are presented for each scenario. We generated six and twelve graphs for Scenario 1 and Scenario 2, respectively. We compared the Frama-C output to the simulation output to determine whether the computed variables were valid. In Scenario 1, we compared the intervals computed by Eva with the deterministic values obtained from simulation data. In Scenario 2, we compared the deterministic values computed by Eva with the deterministic simulation values.

The graphs generated for Scenario 1 plot the variation domain for the output variables of the gyrometer and accelerometer calibration algorithms. Figure 14.9 shows the maximum and minimum values at each time instant for the calibrated accelerometer data on the X-axis (AX). The graph shows that the AX intervals computed by Frama-C contain all simulation values, i.e., there exists no AX value from the simulation that is below the minimum limit or above the maximum limit of the interval computed by Frama-C. We remark that if the Frama-C bounds did not contain the simulation results, this would raise serious concerns regarding the correctness of the case study source code.

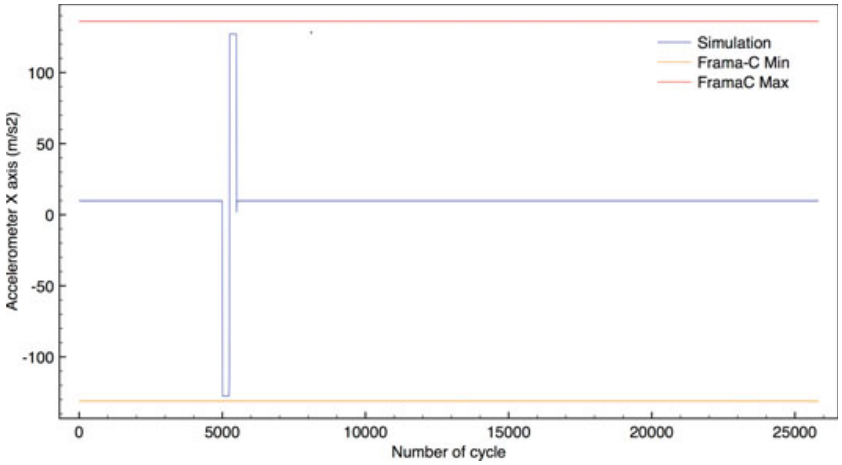

Fig. 14.9 Graph of calibrated accelerometer data on the X-axis in Scenario 1

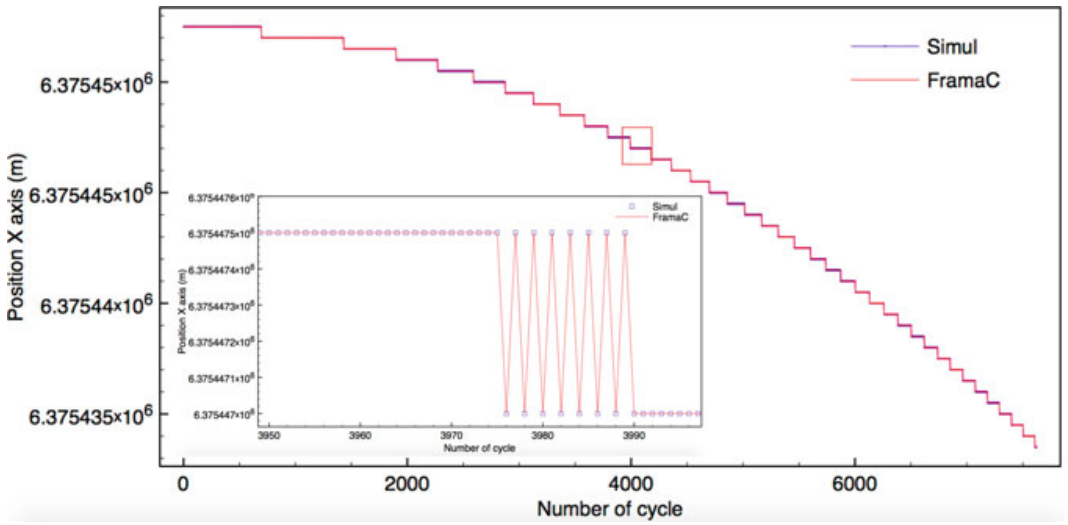

Fig. 14.10 Graph of position data on the X-axis in Scenario 2 with a zoomed-in section

Figures 14.10 and 14.11 show the graphs generated for Scenario 2 regarding the position data on the X-axis and the Euler angles, respectively. Eva produced profiles so similar to those obtained from the simulation that the curves are coincident.

During the implementation and refinement phase, the plug-in issues alarms that represent run-time errors and should thus be recorded as anomalies. In addition to these alarms, as was done in our former experiment as described in Sect. 14.4.1, we applied the same methodology to prepare the case study source code to be executed by Eva. We detected anomalies indirectly during this process in both the context definition and implementation and refinement phases. During implementation, when we found an imprecise or incorrect variable domain, we examined its dependencies in the source code, often finding multiple statements per line related to each dependency. This poor readability made the task of evaluating the domain more difficult. In the

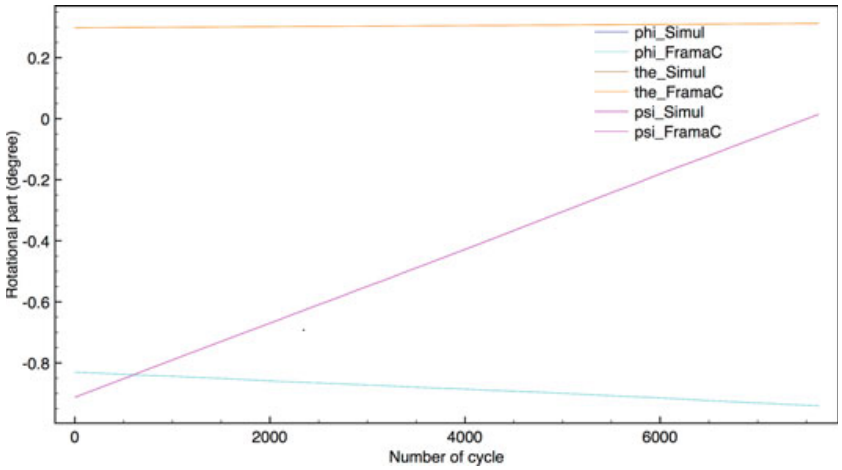

Fig. 14.11 Graph of Euler angles in Scenario 2

documentation, we detected one anomaly in the source code listing of the navigation algorithm, which is attached to a system document. This is presented in Sect. 4.7.

14.4.6.2 SSISNAV Metrics from Eva Coverage Statistics

The common method prescribed by standards to assure the dependability and safety of critical software is to apply measures such as source code statement coverage by test suites [22]. In the context of testing, statement coverage is a measure of the part of the program within which every executable source code statement has been invoked at least once [24].

Eva is currently able to produce semantic coverage metrics. Starting with the Frama-C 19 (Potassium) release, Eva prints an analysis summary outlining the analysis coverage through two measures: the ratio of functions that have been covered by the analysis and the percentage of statements that are reached within these functions. In our interactive approach, the percentage examination is useful for two reasons: it is possible to use these values to confirm if the insertion of a new function or an Eva resource improved the coverage during the analysis, and additionally, this ratio may help in reasoning about statements that are not reached at the end of the analysis.

Tip

The coverage percentages are useful in terms of observing the evolution of the interactive approach, e.g., when a missing function is provided, the coverage increases!

The metrics obtained from Eva and Metrics related to the number of functions analyzed and their reached statements were compared for the preparation phase

Table 14.2 Eva coverage statistics for Scenarios 1 and 2 in the preparation state

	Scenario 1	Scenario 2
Syntactically reachable functions	70 (out of 70)	76 (out of 76)
Semantically reached functions	70	76
Coverage estimation	100.0%	100.0%
Statements reached	1,563 (out of 1,904)	311,478 (out of 311,854)
% statements reached	82%	99%

in Scenario 2. Both plug-ins indicated 100% coverage related to the number of functions analyzed. For the statements reached in these functions, Eva indicated 99% coverage (311,429 statements reached), and Metrics indicated 99.9% coverage (311,478 statements reached). The source code analyzed by Eva was provided as an argument for the Metrics plug-in.

Table 14.2 presents the output of the analysis coverage using Metrics for Scenarios 1 and 2. Scenario 2 presented better results due to the limited dataset in contrast to Scenario 1, where intervals were used. Metrics depends on the results from Eva, but since Metrics itself runs much faster than Eva, the run-times of the Metrics and Eva plug-ins are similar. At the end of the Eva analysis, if any function has not been reached and detecting it by manual verification is difficult, there are two auxiliary options: the Metrics option `-metrics-cover` can identify the functions that are not syntactically reached from the main function, and Callgraph identifies functions that are not called.

The graph in Fig. 14.12 shows the number of functions grouped by the reached statement percentage for the preparation phase in Scenario 2. The unreached function statements depend on our context definition. The unreached statements are located inside branches related to the handling of invalid sensor input data and static memory allocation. This information is only shown by the Frama-C GUI in which it is highlighted.

The functions with the highest coverage mostly contain assignment statements, while the functions with lower coverage contain if—else statements to address error conditions (e.g., alignment functions). In our experiments, we have considered only valid input data obtained from a simulation; therefore, the condition for invalid inputs is never satisfied, which is one of the reasons why analysis does not move into some error-handling branches. Branches that are not analyzed must not be considered formally verified and may contain some errors. In poorly covered functions, it is not assured that the software meets the specified requirements correctly or reliably. In well-covered functions, this assurance is usually better, that is, the probability of guaranteeing that it is error-free is higher. One way to improve the coverage is to create scenarios for addressing non-analyzed functions. In addition, to prevent specific behaviors, the safety team develops new techniques and tools and puts in place new approaches to both system and software engineering. These are all intended to sup-

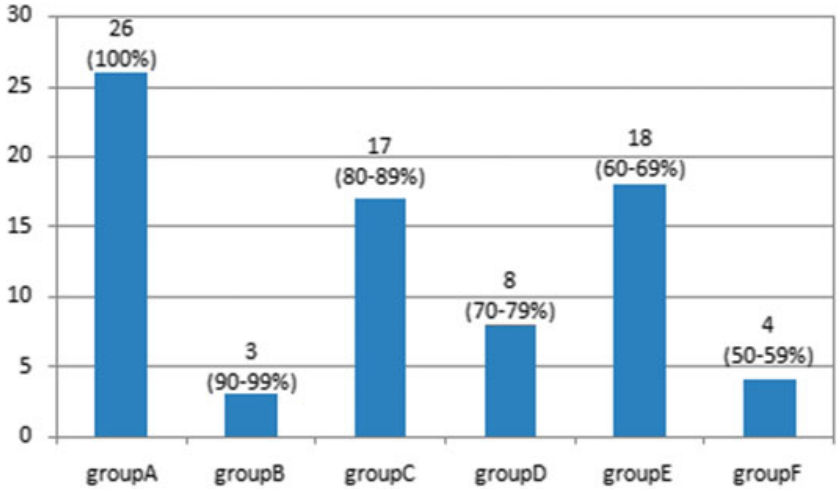

Fig. 14.12 Analyzed function number grouped by reachable statement percentage

port safety analysis, which belongs to the verification activity under the developer's responsibility.

14.4.6.3 SSISNAV Analysis Performance

Considering the high time consumption and cost of the human resources required for formal software verification activity, it is important to measure how time is employed by the analysis, which can be provided by Eva plug-in options.

Option `-eva-show-perf` computes and provides a summary of the time spent analyzing function calls [3] from two perspectives: the first shows the time of all function calls, and the second shows the time considering the sequential order of the function calls.

Option `-eva-flamegraph`, from the Frama-C 14 (Silicon) release, dumps a summary of the time spent to analyze function calls in a format suitable for the flame graph tool [3].

A flame graph is an adjacency diagram with an inverted icicle layout employed to view stack traces [30]. A stack trace is represented as a column of boxes, where each box represents a function. Flame graphs have been adopted by many languages, products, and companies and have become a standard tool for performance analysis [30]. They may help visualize the Eva analysis performance either during execution or later. Interestingly, the flame graph can be generated while the analysis is still running, which is useful when an analysis seems frozen at a certain point but does not display any messages. Running the "frama-c-script flamegraph" creates a flame graph of the probe's current state, while the trace file is still being updated by the probe in progress.

To generate a flame graph while running an analysis, it is sufficient to add the option `-eva-flamegraph` and a name for the output file, as shown in the first

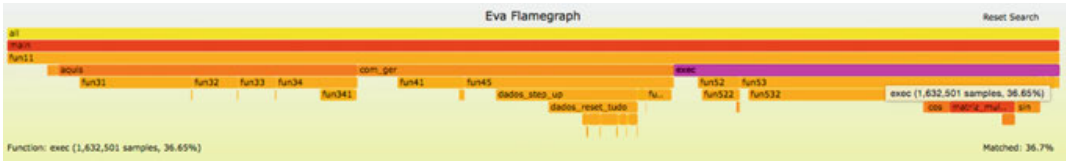

Fig. 14.13 Flame graph of the navigation algorithm for Scenario 2

command below. To visualize a flame graph, Frama-C offers a script to help the user, as shown in the second command below. The command generates the *flamegraph.svg* and *flamegraph.html* files in the default directory (or in the `FRAMAC_SESSION` directory) and opens the html file in a browser [3].

```
$ frama-c -eva file.c -evaflamegraph flamegraph.txt
$ frama-c-script flamegraph flamegraph.txt
```

Figure 14.13 shows the flame graph obtained for the navigation algorithm. The stacks start from the top with the main function (red color), and each called function is stacked underneath. The length of the bar corresponds to the amount of time spent in it. Hovering over a bar shows the function name and the percentage of time spent in it. In the example, clicking on a bar will zoom in on it, increasing the size of its children for better readability. Note that bar colors do not have any special meaning. In addition to hovering, it is possible to search for a function name. The result is a function bar shown in magenta displaying the consumed time percentage.

In addition to the Eva options, it is possible to obtain analysis processing times by using the Unix `time` command. In Scenario 2, the spent times were 108m 18.860 s and 38m 20.947 s in the preparation and navigation states, respectively, while in Scenario 1, they were 79m 56.044 s and 89m 49.306 s, respectively.

14.4.6.4 SSISNAV Analysis Limitations

The case study contains one volatile variable whose analysis results in a wide domain that makes it imprecise. Information concerning this restriction can be found in the Eva manual [11]. A second restriction we faced concerns the maximum number of assignments that are allowed in a function, but this can be easily circumvented by using additional functions. For Scenario 2, a function for storing sensor data in a two-dimensional array [25828][6] exceeded the number of statements that Eva is currently able to process in a single function. To solve this problem, we had to split the function into smaller functions.

14.4.7 Reporting of Anomalies Found in the Analyses

We have considered static analysis as a technique in our software verification process, in the software design & implementation engineering process, and as part of the code

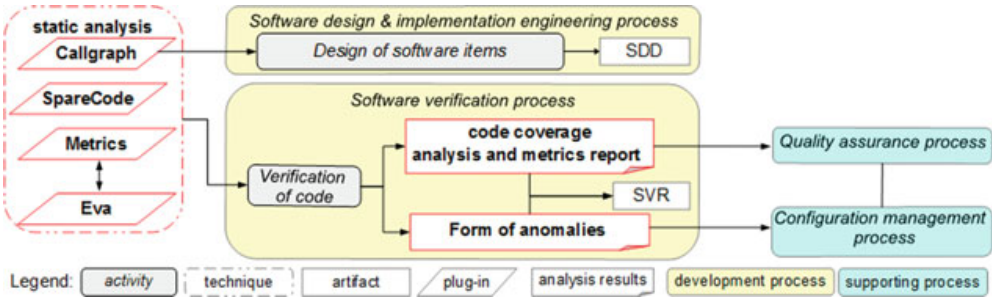

Fig. 14.14 Frama-C plug-ins applied to the software development activities

review activity [15]. After our experiments, we associated the Frama-C plug-ins with the processes and activities in the software development life cycle as proposed by ECSS-E-ST-40C and shown in Fig. 14.14.

In the software design & implementation engineering process, Callgraph can contribute to SDD elaboration because it helps developers by providing an accurate view of function calls. Additionally, the call graphs can be inserted as new figures and/or be compared to other control-flow graphs already present in the SDD.

During the software verification process, the static analysis results can be synthesized as follows: (1) the anomaly form is sent to the configuration management process, which involves discrepancies found in documents, code, and call graphs in the cases of functions that are never called; and (2) code coverage analysis and metrics reports, which are sent to the quality assurance process. Both reports can be useful in the elaboration of the software code verification report since the quality assurance process and the configuration management process, which are responsible for analyzing and approving the results, depend on each other to support the activities in the software development life cycle.

We proposed an anomaly form as shown in Table 14.3, which has been only partially filled in for illustrative purposes. The form fields are based on the IEEE Std 1044–2009 [51] and manual code inspection [46]. Each record of the form contains an anomaly with its attributes, where *asset* and *artifact* are the software asset (product, component, module, etc.) and the specific software work product that contains the anomaly, respectively; *lin/pg* specifies the anomaly location in the artifact, *description* defines the anomaly, *severity* is the degree of impact that the anomaly could cause (inconsequential, minor, major, critical), *suggested corrective action* is the alteration recommended to fix the anomaly, and *plug-in* is the Frama-C plug-in responsible for anomaly detection, either directly or indirectly. The *type* attribute encompasses the anomaly category. We propose the following abridged list of types:

- Function: an anomaly in a function definition, mapping, access, or use, e.g., unused return value of function calls;
- Variable and constant: an anomaly in data definition, initialization, mapping (data traceability), access, or use, e.g., declared but unused variable;
- Program flow control: an anomaly in control statements, e.g., loop variable not properly initialized, incremented and tested;

Table 14.3 Anomaly form with example records

Asset	Artifact	Lin/pg	Type	Description	Severity	Corrective action	Plug-in
Source code	function5a3	243	Comment	Misspelling	Inconsequential	Correct spelling	Eva
Source code	function5a2	137	Program flow control	Accessing index out of bounds	Critical	Correct code	Eva alarm
Source code	function5b4	98	Function	Function not called	Major	Check code	Callgraph
Source code	navigate.c	51	Variable and constant	Unused global variable	Minor	Variable removal	SpareCode
Source code	function5a3	305	Logic	Cyclomatic complexity = 25	Major	Check code	Metrics
Documentation	148–000/B0X	35	Function	Incorrect source code list	Major	Correct documentation	Eva

- Logic: an anomaly in sequencing, branching, or algorithm definition as found in natural language specifications or in implementation language, e.g., excessive branching;
- Defensive programming: an anomaly related to problematic issues that may arise due to unforeseen circumstances, e.g., divisors not tested for zero;
- Unused source code: unnecessary code, such as redundant code, unexecuted code due to a lack of flow control to reach it, code whose results are never used, or even commented code, e.g., code inserted for testing that should have been discarded later;
- Comment: an anomaly in the comments, e.g., incorrect comments that do not contribute to software understanding and maintenance;
- Readability and legibility: an anomaly that makes a program harder to read and understand and more difficult for its elements to be identified, e.g., the presence of multiple statements per line.

Table 14.4 presents a summary in which anomalies are categorized by type and associated with the number of occurrences and the plug-ins responsible for their detection. We remark that the items detected by Eva can occur both in the context definition phase and in the implementation and refinement phase. In the latter phase, the plug-in issues alarms representing errors that occur at run-time. In our case study, these alarms were reported as anomalies and classified as having critical severity. All of the emitted alarms must be corrected before running the analysis with Eva again.

In addition to anomalies, call graphs, code coverage analysis, and metrics reports, the context definition activities, which required a deep program understanding, have contributed to the elaboration of the DD-SRS, detailed DFDs, and DD-SDD.

14.5 Conclusion

The results we obtained indicate that it is a good practice to adopt static analysis techniques during the software verification process because they are capable of guaranteeing greater reliability in the final software product. Since it is not necessary to run the software, static analysis can be used in a step prior to the testing activity, making it possible to start detecting bugs and correcting them even before testing. In addition, it can be a way to validate the software's behavior and to make adjustments before actually starting the testing activity. This can save both money and time because it is easier to modify the software during an initial stage than at later stages of the development life cycle. Implementation efforts are minimized if bugs are found earlier since less reworking is needed.

The adoption of this technique can have a direct impact on other processes in the software development life cycle, including software product assurance, the software maintenance process, and the software design & implementation engineering process. We can say that the obtained benefits are valuable.

Table 14.4 Summary of anomalies related to the source code

Occurrence number	Type	Description	Plug-in
1	Program flow control	Loop variable not properly initialized, incremented and tested	Eva alarm
5	Variable and constant	Uninitialized variable	Eva alarm
2	Defensive programming	Divisor that is untested for zero or incorrect values	Eva alarm
36	Readability and legibility	More than one statement per line	Eva
2	Unused source code	Redundant code	Eva
3	Variables and constants	Equality comparison by the operator == between two floating-point numbers	Eva
17	Unused source code	Commented code, unused or in parts of the program that never runs	Eva
7	Comment	Comment that does not contribute to understanding and maintenance	Eva
12	Function	Return values from function calls that are unused	Eva
2	Variable and constant	Unused variable	SpareCode
3	Variable and constant	Unused variable	kernel
3	Logic	Excessive branching (cyclomatic complexity >20)	Metrics

Static analysis can aid in successfully achieving the very important software property of program correctness through the detection and removal of development errors to build safer, cheaper, and more reliable source code. Analysis code coverage is an important metric in terms of guaranteeing the quality of software products, which is essential for certified and critical software. The higher the percentage of code covered by analysis, the lesser the likelihood that it will contain errors. In general, the definition of acceptable code coverage shall be agreed upon between the customer and the supplier based on the software criticality degree.

Static analysis based on formal methods requires a deep level of understanding of the application to be verified, both in terms of execution and evaluation of the results. Before running an analysis, a study of the application context is required, e.g., which and how the external entities interact with software (input/output data). Furthermore,

the interpretation of results depends on knowledge of the system algorithms and an existing database (e.g., simulation data) to compare the outputs. Software documentation is an important resource to support analysis. This obtained program understanding may help in performing and improving a software maintenance process and, in general, for software engineering processes for new or outsourced software systems of the same type.

The use of static analysis in the software design & implementation engineering process may well affect the coding and software item design activities. One of the most obvious applications of static analysis is in coding activity. It can identify weaknesses in source code that lead to potential vulnerabilities. Additionally, the elaboration of an SDD, an output of the software item design activity, can employ call graphs obtained from static analysis. On the one hand, users can face significant challenges when applying Frama-C. The interaction demanded by some plug-ins, such as Eva and Wp, requires considerable knowledge. Additionally, for experiments involving large volumes of input data, the analysis can be time-consuming.

Despite these challenges, the benefits outweigh the costs. Frama-C is an open-source platform organized into a modular architecture, and its collaborative extensible framework permits developers to implement their own plug-ins and make them available to other users. It is advantageous to have several analyzers in the same framework since the user can perform a variety of analyses using the same tool. Frama-C is regularly updated and effectively maintained by the Frama-C development team, which seeks to meet user requests, as exemplified by our need for a trigonometric function.

Finally, our work has shown that the described development process can be used in the industrial context as well. The integration of Frama-C in the industrial process can be a solution to improve quality by employing static analysis for run-time error detection.

Acknowledgements The authors would like to thank LINCS/IAE, ITA, HASLab, INESC TEC, the University of Minho, and all of the reviewers for their invaluable work on how to improve the chapter. This work is partially financed by the National Funds through the Portuguese funding agency, FCT—Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

References

1. Agência Espacial Brasileira (2022) Transporte Espacial. https://www.gov.br/aeb/pt-br/programa-espacial-brasileiro/transporte-espacial. Accessed 20 Sep 2022
2. Agência Força Aérea: IAE realiza Operação Santa Maria 1/2021 no Centro de Lançamento de Alcântara (CLA) (2021). https://www.fab.mil.br/noticias/mostra/37556. Accessed 24 Jan 2022
3. Alberti M, Antignac T, Barany G et al Help of frama-c tool
4. Alberti M, Antignac T, Barany G et al (2021) FRAMA-C. https://frama-c.com/. Accessed 15 Jan 2022

5. Baudin P, Bobot F, Bühler D, Correnson L, Kirchner F, Kosmatov N, Maroneze A, Perrelle V, Prevosto V, Signoles J, Williams N (2021) The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun ACM. https://doi.org/10.1145/3470569
6. Black PE, Walia KS (2020) SATE VI Ockham sound analysis criteria. Tech. rep., national institute of standards and technology—NIST. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8304.pdf
7. Blanchard A, Kosmatov N, Loulergue F (2018) A lesson on verification of IoT software with Frama-C. In: Conference on high performance computing and simulation (HPCS). https://doi.org/10.1109/HPCS.2018.00018
8. Bonichon R, Yakobowski B Frama-C's metrics plug-in 24.0 (Chromium). https://www.frama-c.com/download/metrics-manual-24.0-Chromium.pdf
9. Brahmi A, Carolus MJ, Delmas D, Essoussi MH, Lacabanne P, Lamiel VM, Randimbivololona F, Souyris J (2020) Industrial use of a safe and efficient formal method based software engineering process in avionics. In: Conference on European congress embedded real time systems (ERTS)
10. Brazilian Space Agency (2022) The Brazilian Space Agency–The bridge to the future. https://www.gov.br/aeb/pt-br/centrais-de-conteudo/publicacoes/LivretoBrazilianSpaceAgency.pdf. Accessed 18 Jan 2022
11. Bühler D, Cuoq P, Yakobowski B, Lemerre M, Maroneze A, Perrelle V, Prevosto V Eva–the evolved value analysis plug-in 24.0 (Chromium). https://www.frama-c.com/download/eva-manual-24.0-Chromium.pdf
12. Bureau Veritas group (2016) Bureau Veritas releases a guide, co-written with the CEA, for enhancing the reliability and performance of embedded software. https://group.bureauveritas.com/fr/node/387. Accessed 27 Sep 2022
13. Carrijo DS, Oliva AP, de Castro Leite Filho W (2002) Hardware-in-loop simulation development. Int J Model Simul. https://doi.org/10.1080/02286203.2002.11442238
14. CEA List (2017) Dassault Aviation innovates in cybersecurity with Frama-C. https://www.cea-tech.fr/cea-tech/english/Pages/ec_2017/dassault-aviation-innovates-in-cybersecurity-with-frama-c-smart-digital-systems.aspx. Accessed 15 Jan 2022
15. Chess B, West J (2007) Secure programming with static analysis. Addison-Wesley
16. Clarke EM, Grumberg O, Peled DA (2000) Model checking. The MIT Press
17. Correnson L, Cuoq P, Kirchner F, Maroneze A, Prevosto V, Puccetti A, Signoles J, Yakobowski B (2021) Frama-C user manual release 24.0 (Chromium). http://frama-c.com/download/user-manual-24.0-Chromium.pdf. Accessed 13 Apr 2023
18. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference on ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL). https://doi.org/10.1145/512950.512973
19. Cuoq P, Monate B, Pacalet A, Prevosto V, Regehr J, Yakobowski B, Yang X (2012) Testing static analyzers with randomly generated programs. In: International symposium on NASA formal methods (NFM). Springer. https://doi.org/10.1007/978-3-642-28891-3_12
20. Duprat S, Lamiel VM, Kirchner F, Correnson L, Delmas D (2016) Spreading static analysis with Frama-C in industrial contexts. In: Conference on European congress embedded real time software and systems (ERTS)
21. Dutle A, Moscato M, Titolo L, Muñoz C, Anderson G, Bobot F (2021) Formal analysis of the compact position reporting algorithm. Form Asp Comput 33(1), 65–86. https://doi.org/10.1007/s00165-019-00504-0
22. ECSS: E-ST-40C space engineering—software (2009)
23. ECSS: Q-HB-80-04A Space product assurance—software metrication programme definition and implementation (2011)
24. ECSS: Q-ST-80C space product assurance—software product assurance (2017)
25. Eisenbarth T, Koschke R, Simon D (2001) Aiding program comprehension by static and dynamic feature analysis. In: IEEE international conference on software maintenance (ICSM). https://doi.org/10.1109/ICSM.2001.972777

26. Frama-C (2012) Frama-C news and ideas homepage, 2015 [Online]. https://frama-c.com/2012/01/16/Csmith-testing.html. Accessed 27 Sep 2022
27. Gansner ER, Koutsofios E, North S (2015) Drawing graphs with *dot*. https://www.graphviz.org/pdf/dotguide.pdf
28. Gao D, Reiter MK, Song D (2004) Gray-box extraction of execution graphs for anomaly detection. In: ACM conference on computer and communications security (CCS). https://doi.org/10.1145/1030083.1030126
29. Gentsch C (2020) Evaluation of open source static analysis security testing (SAST) tools for c. Tech. rep., DLR-German Aerospace Center, Jena. https://elib.dlr.de/133945/
30. Gregg B (2016) The flame graph: this visualization of software execution is a new necessity for performance profiling and debugging. ACM Queue. https://doi.org/10.1145/2927299.2927301
31. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM. https://doi.org/10.1145/363235.363259
32. Hussein M, Nouacer R, Radermacher A (2017) Towards a safe software development environment. In: Euromicro conference on digital system design (DSD). https://doi.org/10.1109/DSD.2017.13
33. Hussein M, Nouacer R, Radermacher A, Puccetti A, Gaston C, Rapin N (2018) An end-to-end framework for safe software development. Microprocess Microsyst. https://doi.org/10.1016/j.micpro.2018.07.004
34. INNOSPACE: INNOSPACE (2023). http://www.innospc.com/myboard/sub04_02. Accessed 13 Apr 2023
35. Instituto de Aeronáutica e Espaço: Projeto SIA (2018). https://www.iae.dcta.mil.br/index.php/todos-os-projetos/todos-os-projetos-desenvolvidos/projetos-sia. Accessed 13 Apr 2023
36. Instituto de Aeronáutica e Espaço: VLS-1 (2019). https://iae.dcta.mil.br/index.php/todos-os-projetos/todos-os-projetos-desenvolvidos/projetos-vls1. Accessed 20 Jan 2022
37. International Organization for Standardization (2017) ISO/IEC/IEEE 24765 Systems and software engineering—Vocabulary
38. Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B (2015) Frama-C: a software analysis perspective. Form Asp Comput. https://doi.org/10.1007/s00165-014-0326-7
39. Kumar N, Neema S, Das M, Mohan BR (2021) Program slicing analysis with KLEE, DIVINE and Frama-C. In: International conference on automation and computing (ICAC). https://doi.org/10.23919/ICAC50006.2021.9594142
40. Maroneze A, Perrelle V, Kirchner F (2019) Advances in usability of formal methods for code verification with Frama-C. Electron Commun EASST. https://doi.org/10.14279/tuj.eceasst.77.1108. Interactive workshop on the industrial application of verification and testing (ETAPS)
41. Ministry of Defense (2012) Defense white paper–Livro Branco de Defesa Nacional. https://www.gov.br/defesa/pt-br/arquivos/estado_e_defesa/livro_branco/lbdna_2013a_inga_net.pdf. Accessed 18 Jan 2022
42. Musco V, Monperrus M, Preux P (2017) A large-scale study of call graph-based impact prediction using mutation testing. Softw Qual J. https://doi.org/10.1007/s11219-016-9332-8
43. NASA (2022) NPR 7150.2D NASA software engineering requirements
44. Ourghanlian A (2015) Evaluation of static analysis tools used to assess software important to nuclear power plant safety. Nucl Eng Technol. https://doi.org/10.1016/j.net.2014.12.009
45. Ramos F (2015) History and current status of SISNAV: a brief report. In: Simpósio Brasileiro de Engenharia Inercial (SBEIN). https://doi.org/10.13140/RG.2.1.3529.0323
46. Romani M, Takahashi P, Lahoz C (2009) A process of code inspection for space software. In: Conf. on Int. astronautical congress
47. RTCA/EUROCAE (2011) RTCA DO-178C software considerations in airborne systems and equipment certification
48. Ryder BG (1979) Constructing the call graph of a program. IEEE Trans Softw Eng. https://doi.org/10.1109/TSE.1979.234183
49. Signoles J (2020) Abstract interpretation and properties of c programs. http://ejcp2019.icube.unistra.fr/slides/js.pdf. Accessed 10 Dec 2020

50. Silva RAB, Arai NN, Burgareli LA, Oliveira JMP, Pinto JS (2016) Formal verification with Frama-C: a case study in the space software domain. IEEE Trans Reliab. https://doi.org/10.1109/TR.2015.2508559
51. Software & Systems Engineering Standards Committee (2009) IEEE 1044 standard classification for software anomalies
52. de Souza J, Filho W (2012) Sistema de Navegação Inercial SISNAV - Mecânica e Eletrônica Embarcada. In: Simpósio Brasileiro de Engenharia Inercial (SBEIN)
53. Stack Overflow (2019) Sparecode analysis in Frama-C. https://stackoverflow.com/questions/59240081/sparecode-analysis-in-frama-c
54. TrustInSoft (2022) https://trust-in-soft.com. Accessed 15 Jan 2022
55. VESSEDIA (2022) https://cordis.europa.eu/project/id/731453. Accessed 15 Jan 2022