Using Abstract Interpretation to Produce Dependable Aerospace Control Software

Rovedy Aparecida Busquim e Silva, rovedyrabs@iae.cta.br
Nanci Naomi Arai, nancinna@iae.cta.br
Luciana Akemi Burgareli, lucianalab@iae.cta.br
Laboratorio de Engenharia de Software / Instituto de Aeronautica e Espaco (IAE), Brasil

Jose M. Parente Oliveira, jparente@ita.br
Divisao de Ciencia de Computacao / Instituto Tecnologico de Aeronautica (ITA), Brasil

Jorge Sousa Pinto, jsp@di.uminho.pt
HASLab / INESC TEC & Universidade do Minho, Portugal

Abstract

In the context of software dependability, the software verification process has an important role. Formal verification of programs is an activity that can be inserted in this process to improve software reliability. This paper presents the definition of an approach that employs a formal verification technique based on abstract interpretation. The main goal is to apply this technique as a formal activity in the software verification process to help software engineers identify programs faults. The applicability of the proposed approach is demonstrated by a case study based on embedded aerospace control software. The results obtained from its use show that abstract interpretation can contribute to software dependability.

1. Introduction

Accidents caused by errors in software are a reality in critical real-time systems. To find faults throughout the software development life cycle and develop dependable software, software verification and validation processes must be used. Formal methods, complementary to traditional software verification and validation methods, are recommended in standards specifically used in the aerospace area such as ECSS-E-ST-40C [9].

The formal verification of programs based on abstract interpretation is the focus of this work. The goal is to complement the software testing and simulation activities. The testing activity may provide a partial verification of the software behavior using a subset of selected input data. A test-generation tool would necessarily pick only a subset of the billions of possible entries to execute the application [7]. On the other hand, the results given by a tool based on abstract interpretation are valid for all the range of values proposed by the user. In this context, the approach presented here proposes the application of abstract interpretation to verify embedded aerospace control software by the Frama-C static analyzer [7].

The paper is organized as follows. Section 2 presents briefly the background in formal methods, formal verification of programs, and abstract interpretation. Section 3 describes the verification approach. Section 4 shows the practical experience. Section 5 presents related works. Finally, Section 6 is the conclusion, with a discussion about specific issues and future work.

2. Background

2.1. Formal Methods

Formal methods are abstract interpretations, which differ in the way the abstract semantics is obtained [4]:

- model-checking, where the abstract semantics is obtained manually by the user in the form of a finitary model of the program execution;
- deductive methods, where the abstract semantics is specified by verification conditions by the user in the form of inductive properties that satisfy (true at each program step, such as loop invariants) these verification conditions;
- static analysis, where the abstract semantics is computed automatically using the predefined approximations possibly manually parameterizable by the user.

2.2. Formal Verification of Programs

Formal verification of a program is related to the use of formal methods at the level of implementation of programs. In general, an implementation of program is divided into syntax and semantics around the programming languages. Most often in the software development process, the developers are concerned with the syntax of a program, and the semantics is implicitly considered. The main focus is to build an executable object source, and the correction of the program is obtained through testing and simulation. However, in the context of formal verification of a program, the semantics has an important role to play in the development of software applications because it is concerned with rigorously specifying the meaning or behavior of programs. A way to perform formal verification of programs is to use static analysis based on abstract interpretation. A static analyzer based on abstract interpretation is built by a rigorous mathematical theory.

2.3. Abstract Interpretation

Abstract interpretation considers abstract semantics, which is a superset of the concrete program semantics. Concrete program semantics is the most precise semantics that very closely describes the actual execution of the program and formalizes the set of all possible executions of this program in all possible execution environments [4].

Abstract interpretation can be understood through two main views. The first encompasses a semantic to depict the abstract information (derived from concrete semantic) and an abstract function to relate the abstract to the concrete information. The second encompasses an algorithm to evaluate the semantic information interactively [3].

The basic idea of abstract interpretation is to transform the program from a concrete domain to an abstract domain to infer results without running the program. Thus, a static analysis technique relates abstract analysis to executions of the program [5].

A static analysis tool based on abstract interpretation is built by combining abstract domains. It applies an abstract transformer in the program, and an over-approximation of the least fix point of this transformer is computed iteratively by extrapolation operators such as *widening* and *narrowing* [1].

According to Bouissou [2], static analysis by abstract interpretation has been very successful in automatically verifying complex properties of real-time, safety-critical embedded systems.

3. Proposed Verification Approach

3.1 Tool

Frama-C is a platform dedicated to the static analysis of source code written in C constructed as plug-ins. The plug-in of interest is the Value Analysis, which is based on abstract interpretation, and automatically computes sets of possible values for the variables of the analyzed program [7]. The Value Analysis/Frama-C can be useful to detect bugs, prove their absence, and get familiar with foreign code. It is most useful for embedded code because it does not demand dynamic allocation and uses few functions from external libraries [7].

3.2 Approach

The verification approach is based on the following steps:

1. Identifying an analysis context for the case study;
2. Adapting the source code to conform to Value Analysis/Frama-C execution;
3. Running the analysis and refining the results;
4. Addressing the alarms.

The first step is to identify the analysis context. It is necessary to define an entry point for the analysis. The software documentation is essential to conduct this step.

In the second step, it is necessary to adapt the source code: verifying libraries related to the Real Time Operational System (RTOS) and the hardware access as well as the mathematical functions to detect missing functions that must be provided by the appropriate source code.

In the third step, the idea is to refine the analyzer results. Because static analyzers based on abstract interpretation are always sound and always terminate, they are necessarily incomplete [1]. Therefore, false alarms can occur. To improve the results, arguments can be passed to the analyzer; however, this will cost more time to determine whether each alarm is true or false.

The purpose of the fourth step, is to address the alarms. The process consists of picking an alarm, investigating its causes, watching variables values, and keeping track of the anomalies encountered as well as the code changes. It could be necessary to interact with the developers when severe anomalies are suspected or their causes are doubted.

4 Case Study

The case study is based on a spacecraft with four stages capable of launching satellites weighing maximum 350 kg

at altitudes up to 1000 km [10]. Its embedded software performs the initiation, verification, and control of the vehicle. The source code consists of approximately 15 KLOC of C language distributed in 120 files. The selected part of source code is related to the flight control task, which is the most critical part of the code. For simplification issues, only functions related to the first stage were analyzed. The main functions in the first stage are navigation and event control algorithms. The case study has 12306 SLOC, 47 statements, 12155 if assignments, 9 loops, 50 calls, 2 gotos, and 3 pointer access. This data are results produced by the Metrics/Frama-C plugin.

The case study considered two scenarios where the sensor measurements were treated in different ways. The first scenario considered the maximum range of values accepted by the sensors. In the second scenario, a different approach was adopted. All the sensors measurement values obtained from a flight simulation were used as input to the Value Analysis/Frama-C.

Following the steps of the proposed verification approach, the first activity was to prepare the analysis context. The main function, which is the entry point for the analysis, was created. In this step, the Software Requirements Specification (SRS), Data Dictionary (DD), and Software Design Document (SDD) were essential for looking up information. The logic model in the SRS, represented by Data Flow Diagram (DFD), helped identify the input data for the functions. For example, the input for the navigation algorithm is the flight nominal data and the sensor values that must be provided.

Following the second step, the first activity was to address the include files. The method used was to exclude all the include files of the source code and to include them one by one in the next executions as required. This way, it was possible to detect a number of include files that were unnecessary. Then, the missing functions were addressed. For the RTOS libraries, an implementation of functions was provided with similar behaviors to the real ones. For the sensors measurements, the analysis was parameterized adding non-determinism through the *Frama_C_float_interval* and *Frama_C_interval* functions. For the hardware output, functions were created to examine the intermediate results through the tool functions. For mathematical functions, functions from the Frama-C itself were used when available or functions with similar behaviors to the original ones were implemented. After that, it was possible to run the verification, but the first results were imprecise as expected because no refinement was made. Becoming familiar with the source code was a complementary result of this step.

The next step was to make the verification more precise. Frama-C asserts, parameters, and functions were studied and used to refine the analysis, address some false alarms, and improve the results. In the analyzed case study, the various large loops and several control flow statements required special attention to be dealt with. Options were used as *-slevel* for unrolling loops and control flow statements to obtain more accurate results. However, these options usually slow down the analysis execution; therefore, it was necessary to strike a balance between the required precision and the time spent by the analysis.

In the last step, three warnings related to float operation were found. By examining the function, it was possible to ignore them. This is because the analyzed source code is related to control algorithms, software with floating-point computations. These kinds of messages appear when a floating-point operation could result in an infinite value or *Not a Number* (NaN).

The main proposal of the Value Analysis/Frama-C is to provide valid results for all the range of values proposed by the user, as opposed to what a test-generation tool would typically do.

The first scenario is often a useful method for using the Value Analysis/Frama-C. In this case, the analysis was performed smoothly and provided valid ranges of values for the variables. As a conclusion, it was possible to infer that the implemented algorithms are working properly without any bugs, divisions by zero, invalid pointer access, buffer overflows, and other run-time errors.

In the second scenario, Value Analysis/Frama-C worked as C interpreter [6]. As a result, the analysis gave variables profiles very similar to the simulation. To compare them, three linear velocity parameters (Vx,Vy,Vz) were chosen, and the plots are shown in the Figure 1. The results obtained from the flight simulation and the Value Analysis/Frama-C are so alike that the plots are indistinguishable, and in spite of having six plots, it is possible to observe only three.

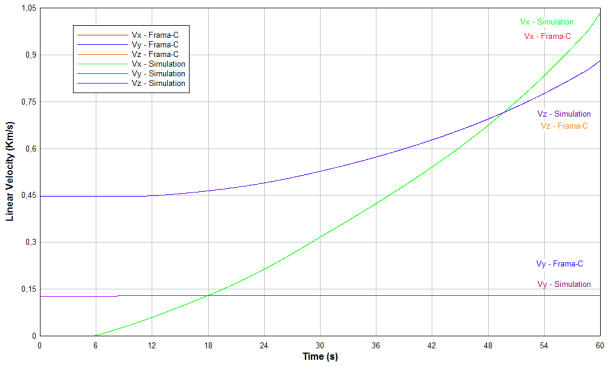

Figure 1. Values computed by Frama-C x Simulation data.

5 Related Work

This section presents research works related to formal verification of programs. Vijay makes a survey of automated techniques for Formal Software Verification and concludes that, instead of the false alarms emitted, the static analysis techniques based on abstract interpretation scale well at the cost of limited precision [8]. In an experiment on static analysis of the XEN Kernel, Pucetti [11] concludes that abstract interpretation is the most promising technique to extract runtime level bugs from the code with little user assistance. Bouissou [2] shows the results of an ESA funded project on the use of abstract interpretation to validate critical real-time embedded space software. In this paper, two tools are presented: a static program analyzer (ASTREE) and an abstract interpretation tool for studying numerical programs coded in C (FLUCTUAT). The conclusion is that the performance of this new generation of tools has dramatically increased.

6. Conclusions, Discussion, and Future Work

The presented work applies a static analysis tool based on abstract interpretation for formal verification on embedded aerospace control software. It has advantages compared to the other two fundamental formal approaches described in the section 2.1. When compared to deductive methods, the proposed approach has the advantage of automation, because it is not necessary to include so many annotations in the source code. In relation to the model checking, the static analysis can avoid the state space explosion problem.

Some results obtained are worth being discussed. In the analysis context, it was possible to detect an error related to the domain of sensor values in the Data Dictionary. The recommendation is to review the document. Several inclusions were detected of unnecessary include files. Considering that it is an embedded source code, the best approach is to maintain only the necessary source code. Therefore, the recommendation is to remove the unnecessary include files from the source code. The analysis detected global parameters being passed by value in a function related to the sensor measurements. The recommendation is to delete parameters and to perform the analysis of global variable usage.

The main benefit of the proposed approach is to contribute to the development of dependable software systems through the application of formal methods to remove software faults. Removing software faults directly improves dependability because they are no longer potential causes of failure. Adding formalism in the dependability analysis will increase the software quality and decrease the probability of a lost mission in the aerospace field. Of course, this approach alone is not appropriate. It must be a complementary activity in the software process verification, which help to reduce cost in the software validation process.

Future works could include exploring the Frama-C plug-in related with the deductive verification working together Value Analysis.

Acknowledgments

This research was sponsored by Agencia Espacial Brasileira (AEB). This work is partially funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundacao para a Ciencia e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020486.

References

[1] 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. In *AIAA Infotech@Aerospace 2010*, number AIAA-2010-3385, pages 1–38. American Institute of Aeronautics and Astronautics, April 2010.

[2] 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 *In Proceedings of the International Space System Engineering Conference, Data Systems in Aerospace (DASIA 2009)*, volume SP-669, pages 1–7, Istambul, Turkey, May 2009. ESA.

[3] A. Cheng. Abstract Interpretation Webpage. http://www.eleceng.adelaide.edu.au/personal/acheng/public/absInt/absIntMain.html, 2012.

[4] P. Cousot. Abstract Interpretation in a Nutshell. http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, volume 82, pages 238–252, Los Angeles, California, New York, September 1977. ACM Press.

[6] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer Berlin Heidelberg, 2012.

[7] P. Cuoq and V. Prevosto. Frama-C's value analysis plug-in 20110201 carbon version, 2010.

[8] V. D'Silva, D. Kroeningn, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[9] ECSS. ECSS-E-ST-40C Space Engineering - Software, March 2009.

[10] IAE. Instituto de Aeronautica e Espaco - Projeto VLS. http://www.iae.cta.br/?action=vls, 2011.

[11] A. Puccetti. Static Analysis of the XEN kernel using Frama-C. *Journal of Universal Computer Science*, 16(4):543–553, 2010.