Electronic Communications of the EASST
Volume 77 (2019)

Interactive Workshop on the Industrial Application of Verification and Testing, ETAPS 2019 Workshop (InterAVT 2019)

Advances in Usability of Formal Methods for Code Verification with Frama-C

André Maroneze, Valentin Perrelle and Florent Kirchner

6 pages

Guest Editors: Anila Mjeda, Stylianos Basagiannis, Goetz Botterweck
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Advances in Usability of Formal Methods for Code Verification with Frama-C

André Maroneze, Valentin Perrelle and Florent Kirchner*

firstname.lastname@cea.fr
CEA, List, Gif-sur-Yvette, France

Abstract: Industrial usage of code analysis tools based on semantic analysis, such as the FRAMA-C platform, poses several challenges, from the setup of analyses to the exploitation of their results. In this paper, we discuss two of these challenges. First, such analyses require detailed information about the code structure and the build process, which are often not documented, being part of the implicit build chain used by the developers. Unlike heuristics-based tools, which can deal with incomplete information, semantics-based tools require stubs or specifications for external library functions, compiler builtins, non-standard extensions, etc. Setting up a new analysis has a high cost, which precludes industrial users from trying such tools, since the return on investment is not clear in advance: the analysis may reveal itself of little use w.r.t. the invested time. Improving the usability of this first step is essential for the widespread adoption of formal methods in software development. A second aspect that is essential for successful analyses is understanding the data and navigating it. Visualizing data and rendering it in an interactive manner allows users to considerably speed up the process of refining the analysis results. We present some approaches to both of these issues, derived from experience with code bases given by industrial partners.

Keywords: semantic analysis, code analysis, visualization

1 Introduction

While formals methods have gained in maturity during the last decades, they struggle to impose in standard industrial processes. The efforts needed to get verification tools running on a general purpose software are demanding, and the cost is often prohibitive. However, with the continuous increase in software complexity, its widespread application, and emerging safety and security threats, the stakes for code verification are getting higher than ever.

Current approaches are currently focused on verifying safety-critical code with a well-delimited perimeter, e.g. without dependencies on external libraries. They are primarily used by teams which can invest a substantial amount of time in them, and often rely on the presence of formal methods experts. To reach a wider range of applications, there are countless obstacles to overcome. The question then stands: what types of improvements can help formal verification apply to general-purpose code?

* This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 830892 for project SPARTA.

FRAMA-C is an open-source software analysis platform [KKP+15] that implements several formal methods-based approaches to perform code verification on real-world C programs. In particular, FRAMA-C includes plugins for abstract interpretation [BBY17], weakest-preconditions computation [Cor14], model-checking [SP16], invariant generation [OPHB18], and runtime verification [SKV17]. To this day, FRAMA-C counts more than 40 known plugins, with about half of them being distributed at frama-c.com/download.html. To try to answer the above question, our approach is to iteratively expand the FRAMA-C toolkit to address each individual problem with easy-to-prototype solutions. We design these solutions while experimenting the analysis process on a wide variety of codes, from industrial applications [Our15, CDDM12] to common open-source case studies [OSC]. This experience report provides some insights on this process and its results; while focused on the use of analyses rooted in abstract interpretation, it keeps a mind toward conclusions that can apply to other types of formal verification.

We begin this paper with a short description of how a typical analysis is conducted with our tool in section 2. Section 3 describes a few recurring problems during the setup of analyses and the solutions we gave to these problems. Once an analysis has been completed, the user must often face a list of several hundreds of alarms, some of them spurious. Section 4 explores the ways to overcome these alarms. Finally, section 5 discusses possible threats to the validity of our conclusions.

2 On the anatomy of a formal code analysis

A formal code analysis involves not only the source code itself, but also the build setup (especially for low-level languages such as C/C++), the target environment (architecture, compiler and runtime) and external libraries. The initial setup of an analysis consists of retrieving this information, often available in the build chain or development environment, and exposing it to the analysis tool.

After the initial setup, the configuration phase largely consists in obtaining information for libraries used in the code: standard language libraries, compiler built-ins, or third-party code libraries. Providing the entirety of this code is often impossible (if the code is not available, e.g. being hardcoded in the compiler or runtime) or impracticable (e.g. too large). A more efficient approach consists in providing information on-demand, only for functions which are effectively reachable in the program under analysis. This information can be a stub – a short piece of code emulating the actual behavior, in a simplified or abstract way – or an abstract specification written in a specification language supported by the semantic analyzer. The on-demand approach means that, whenever new code is analyzed, one must check whether the used functions already have stubs/specifications, or new ones must be written. This can be part of a collaborative effort, incrementally improving support for common libraries.

Once setup is done, we enter an iterative process: an analysis is performed, and either new library functions are called, requiring more stubbing or specification effort, or a more refined parametrization of the analysis is obtained. The end result is usually a list of alarms, warnings, and recommendations, and in the case of sound tools, the guarantee of absence of certain alarms and the proof of certain properties. This iterative process usually requires more computation time but is less user-intensive, thus several analyses can be run in parallel. Analyses typically begin

focused on efficiency (to ensure quick termination) and are then refined towards longer but more precise results. The non-linearity of the process favors the use of profilers to understand which functions are taking most of the time. Low-cost profiling can be obtained by integrating existing tools, such as flamegraphs [fla].

3 Smarter, faster code analysis setups

As mentioned previously, the initial steps of an analysis require an intensive setup effort, compared to the later steps. Even though this initial part has the advantage that results can often be reused later, for the end user who is interested only in analyzing their source code, the short term cost is hardly justifiable. Minimizing it is essential to allow semantic tools to be more widely experimented with.

Practical experience with the setup of new code bases for analyses, both coming from open source projects and proprietary industrial code, identified the following challenges:

1. **Identifying and retrieving build commands.** Languages such as C/C++ contain a lot of complexity in their build systems: the C preprocessor uses information not present in the source code, such as macros and inclusion directories specified via the command line. This information is usually defined once and stored in the IDE (e.g. Visual Studio) or in build files (Makefile, CMakeLists.txt). It is necessary for most analysis tools, since it is a prerequisite for parsing. The LLVM project defined a JSON Compilation Database format [JCD], under the form of `compile_commands.json` files which contain the list of commands (including all arguments) passed to the compiler during a build. Verification tools can parse it to obtain the necessary flags. CMake and `bear` (Build EAR) [BEA], a wrapper for Makefiles that intercepts compiler calls, are some tools which able to produce these databases. For IDEs which store this information in a different format (e.g., XML files with project metadata such as Visual Studio's `.vcxproj` files), conversion tools can be devised to extract the data.

2. **Source identification.** Another requirement for some semantic tools is the notion of a *whole program*: in order to analyze as deeply as possible, source information must be complete: all sources should be given at once. However, simply concatenating the list of source files is not always feasible. For instance, the source code of the SQLite library contains 28 applications in the `tool` directory, 11 applications in the `test` directory, plus several others, all of which can serve as entry points for a semantic analysis and which redefine the `main` function. Also, external libraries are typically not included in the compilation, except for their headers. Transforming the command line to ensure all information is present, without duplication, requires some extra tooling. A cheap solution to this issue is an iterative try-and-fail approach, in which the analysis starts with the minimal amount of sources, and whenever a missing definition is found to be reachable, its source is added to the compilation. User interaction is often required to decide between multiple choices.

3. **Analysis parametrization based on templates.** Semantic analyses can be configured along several trade-offs between precision and efficiency, as well as several hypotheses

concerning the code base and execution environment (can memory allocation fail? Are floating-point NaNs expected? Should padding bits be considered initialized memory?, etc.). Extensive `help` commands, user manuals and tutorials provide a multitude of information, but the user might not have time to read everything before trying the tool. Default values must be provided for the common case, but the very large range of parametrization options available in such tools (as large as, if not larger, than what compilers provide) is better suited via the definition of templates coupled with user feedback to provide the most specific set of parameters relevant for the analysis.

4 Exploiting code analysis results

Code analyses are typically able to report dozens if not hundreds of issues, which often drive the user away, due to the amount of work they seem to imply. Prioritization of warnings and alarms is one of the essential features that semantic analyzers must provide to their users. However, when it is not possible to define such priorities, other means must be provided to the user. One important feature that has not been observed in current tools is the visualization of stack traces and analysis contexts, in a scalable way (to be able to handle the thousands of possibles paths present in existing code bases) while also providing useful insight to the user as to the origin of alarms, and which measures are more likely to reduce the number of false alarms.

Origin of alarms. The identification of the origin of alarms is related to *program slicing*: highlighting the statements which the alarm depends on allows the user to focus on a smaller subset of the program. Interprocedural aspects must be taken into account: filtering of callstacks, code navigation between functions, and the kinds of dependencies related to the alarm: data dependencies, control dependencies, and analysis-specific dependencies (e.g. abstract analyses might include over-approximations as possible causes of alarms). Some of these sources are more amenable to textual descriptions than others; most analysis tools include graphical interfaces with code display features allowing for natural mappings (e.g. code highlighters and gutter indicators for statements, filetree viewers for callstacks and interprocedural dependencies). However, complex analyses involve a multitude of call contexts and variable values, leading to information overflow that requires advanced filtering capabilities to ensure it remains manageable by the user.

Reduction of false alarms. Semantic analyses resorting to approximations (either to avoid missing potential alarms, or to improve the efficiency of the analysis) may produce false alarms. The information associated with these alarms, as well as the way they are displayed, allow the user to more efficiently tune the analysis and add assertions to eliminate them. For instance, if a given function contains too many alarms, the user may decide to stub it with a more abstract version or specification. Clustering alarms by locality (either temporal or spatial) may help identify common causes. Interactive dependence graphs, with an initial overview and on-demand zooming and unfolding of details, offer a way to present hierarchical information without overloading the user. The extra effort spent in presenting these graphs is compensated in the long term by the time savings in understanding the analysis results.

5 Threats to validity

The challenges presented here come from our experience as direct users of the semantic analyses available in FRAMA-C, as well as from feedback from users of the platform, several of which are formal method practitioners. Feedback is not necessarily representative of the developer community at large. The solutions mentioned in this paper, some of them in preliminary stages, were tested mostly by ourselves, with a few of them having been tested by platform users at large. Usability improvements are continuously deployed in the development version, but there might be a long delay between their introduction and external user feedback about their efficiency. Also, many of the issues concern difficulties present in C/C++ (e.g. preprocessor), but not always as prominent in other languages.

6 Conclusion

Semantic analysis tools, due to their complexity and the amount of trade-offs they allow, need better usability to enable a more widespread adoption. The integration of existing tools, automatization of the initial setup, and use of templates complemented with user feedback are first steps in this direction. Collaborative development of semantic annotation/stubbing for standard libraries should improve the applicability of such analyses towards more code bases. Templates with user feedback, graph visualization for understanding of analysis results and profiling are all complementary techniques which offer improved ease of use at a low development cost. To incentivize application of semantic tools, formal methods scientists and engineers can use open source code bases both as source of input data for future development of the analysis, as well as examples of parametrizations to be reused in other codes.

Bibliography

[BBY17] S. Blazy, D. Bühler, B. Yakobowski. Structuring Abstract Interpreters Through State and Value Abstractions. In Bouajjani and Monniaux (eds.), *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*. Lecture Notes in Computer Science 10145, pp. 112–130. Springer, 2017.
doi:10.1007/978-3-319-52234-0_7
https://doi.org/10.1007/978-3-319-52234-0_7

[BEA] Build EAR Github Page. https://github.com/rizsotto/Bear. Accessed: 2019-01-25.

[CDDM12] P. Cuoq, D. Delmas, S. Duprat, V. Moya Lamiel. Fan-C, a Frama-C plug-in for data flow verification. In *Embedded Real Time Software and Systems (ERTS'12)*. 2012.

[Cor14] L. Correnson. Qed. Computing What Remains to Be Proved. In Badger and Rozier (eds.), *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*. Lecture Notes in Computer Science 8430, pp. 215–229. Springer, 2014.

doi:10.1007/978-3-319-06200-6_17
https://doi.org/10.1007/978-3-319-06200-6_17

[fla] Flame Graphs. http://www.brendangregg.com/flamegraphs.html. Accessed: 2019-01-25.

[JCD] JSON Compilation Database Format Specification. https://clang.llvm.org/docs/JSONCompilationDatabase.html. Accessed: 2019-01-25.

[KKP+15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27(3):573–609, 2015.
doi:10.1007/s00165-014-0326-7
https://doi.org/10.1007/s00165-014-0326-7

[OPHB18] S. de Oliveira, V. Prevosto, P. Habermehl, S. Bensalem. Left-Eigenvectors Are Certificates of the Orbit Problem. In Potapov and Reynier (eds.), *Reachability Problems - 12th International Conference, RP 2018, Marseille, France, September 24-26, 2018, Proceedings*. Lecture Notes in Computer Science 11123, pp. 30–44. Springer, 2018.
doi:10.1007/978-3-030-00250-3_3
https://doi.org/10.1007/978-3-030-00250-3_3

[OSC] Open Source Case Studies Github Page. https://github.com/Frama-C/open-source-case-studies. Accessed: 2019-01-25.

[Our15] A. Ourghanlian. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering and Technology* 47(2):212 – 218, 2015. Special Issue on ISOFIC/ISSNP2014.
doi:https://doi.org/10.1016/j.net.2014.12.009
http://www.sciencedirect.com/science/article/pii/S1738573315000091

[SKV17] J. Signoles, N. Kosmatov, K. Vorobyov. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper). In Reger and Havelund (eds.), *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*. Kalpa Publications in Computing 3, pp. 164–173. EasyChair, 2017.
http://www.easychair.org/publications/paper/t6tV

[SP16] S. Shankar, G. Pajela. A Tool Integrating Model Checking into a C Verification Toolset. In Bosnacki and Wijs (eds.), *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*. Lecture Notes in Computer Science 9641, pp. 214–224. Springer, 2016.
doi:10.1007/978-3-319-32582-8_15
https://doi.org/10.1007/978-3-319-32582-8_15