Formal Verification, Engineering and Business Value

Ralf Huuck
NICTA
Sydney, Australia
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
`ralf.huuck@nicta.com.au`

How to apply automated verification technology such as model checking and static program analysis to millions of lines of embedded C/C++ code? How to package this technology in a way that it can be used by software developers and engineers, who might have no background in formal verification? And how to convince business managers to actually pay for such a software? This work addresses a number of those questions. Based on our own experience on developing and distributing the Goanna source code analyzer for detecting software bugs and security vulnerabilities in C/C++ code, we explain the underlying technology of model checking, static analysis and SMT solving, steps involved in creating industrial-proof tools.

1 Motivation

Formal verification has come a long way from being a niche domain for mathematicians and logicians to an accepted practice, at least in academia, and frequently being taught in undergraduate courses. Moreover, starting out from a pen-and-paper approach, a range of supporting software tools have been developed over time including specification tools for (semi-)formal languages such as UML, Z or various process algebrae, interactive theorem-provers for formal specification, proof-generation and verification, as well as a large number of algorithmic software tools for model checking, run-time verification, static analysis and SMT solving to name a few [5].

Despite all the effort, however, there has been only limited penetration of verification tools into industrial environments, mostly remaining confined to the respective R&D laboratories of larger corporations, defense projects or selective avionics work. The use of verification tools by the average software engineer is rare and typically stops at formal techniques built into the compiler or debugger.

One of the key motivations for our work has been to contribute to some of the technology transfer from classical academic domains to industrial applications and use. In particular, we have been working on bringing verification techniques such as model checking [8, 2], abstract interpretation [3] and SMT solving [4] to professional software engineers.

To make verification technology applicable, we believe a number of core principles must be met: First of all, any verification tool has to be so simple to use that it does not require much or any learning from users outside the formal methods domain. Secondly, the performance of the tool has to match the typical workflow of the end-user. This means, if the end-user is accustomed to doing things in a particular order, those steps should remain largely the same. Moreover, run-time performance of any additional analysis or verification should be similar to existing processes. Finally, and most importantly, a new analysis tool should provide real value to an end-user. This means, it should deliver information or a degree of reliability that was previously not available, making the use of the tool worthwhile.

2 Software Tool Challenges

The result of our endeavor is *Goanna* [6], an automated software analysis tool for detecting software bugs, code anomalies and security vulnerabilities in C/C++ source code. Goanna is designed to be run at compile-time and does not require any annotations, code modifications or user interaction. Moreover, the tool can directly be integrated into common development environments such as Eclipse, Visual Studio or build systems based on, e.g., Makefiles. To achieve acceptance in industry, all formal techniques are hidden behind a typical programmer's interface, all of C/C++ is accepted as input (even, e.g., Microsoft specific compiler extensions) and scalability had to be ensured for many millions of lines of code in reasonable time.

To achieve this, a number of trade-offs had to be made: While using a range of formal verification techniques, Goanna is not a verification tool as such, but rather a bug detection tool. This means, it does no conclusively show the absence of errors, but does its best to find certain classes of bugs. Moreover, Goanna comes by default with a fixed set of pre-defined checks for errors such as buffer overruns, memory leaks, NULL-pointer dereferences, arithmetic errors, or C++ copy control mistakes as well as with support for certain safety-critical coding standards such as MISRA [1] or CERT [9] totaling over 200 individual checks. To achieve reasonable performance that is of the same order as the compiler, a number of assumptions and approximations (as well as refinements) are made. Naturally, this leads to missed errors (false negatives) as well as to spurious errors (false positives). Finding the right balance between precision, speed and number of supported checks is very much an engineering art supported by new verification and abstraction techniques. This work will focus on some key insights:

Core Analysis The core of our program analysis is based on CTL model checking. In particular, most static analysis tasks such as NULL-pointer detection, buffer overrun analysis and memory leak detection are outsourced to an explicit state CTL model checker and an abstract interpretation framework. The approach of model checking the distribution of syntactic elements in C/C++ code creates particular requirements such as handling many small verification tasks. We implemented our own explicit state model checker outperforming some existing state-of-the-art implementation by up to an order of magnitude.

Refinements & Heuristics Goanna, in its academic version, integrates with SMT solvers and applies a refinement loop for distinguishing spurious errors from real ones [7]. Moreover, a substantial engineering effort has been done to apply heuristics matching common programmer patterns and expectations to keep the overall false-alarm rate to a minimum.

Engineering Challenges While formal methods tend to be precise and define right and wrong, software practice is often much less rigorous and one person's bug is another person's feature. Moreover, C/C++ are complex languages with often tricky semantics and, more importantly, are neither stable with new features being added through the C++11 standard or compiler extensions, nor are these languages the same for everyone, but rather determined by how the individual compiler manufacturer supports and interprets C/C++.

Internally, we have to deal with huge number of verification tasks. This can range to hundreds of millions of model checking specifications for larger projects and sometimes billions of pre-processing pattern matching queries. This is typically contrary to an academic environment, where one is often interested in a few complex verification tasks only. Apart from new algorithms this requires dedicated engineering solutions for efficient caching, incremental analysis, inter-function summaries and databases for storage and look-up between runs.

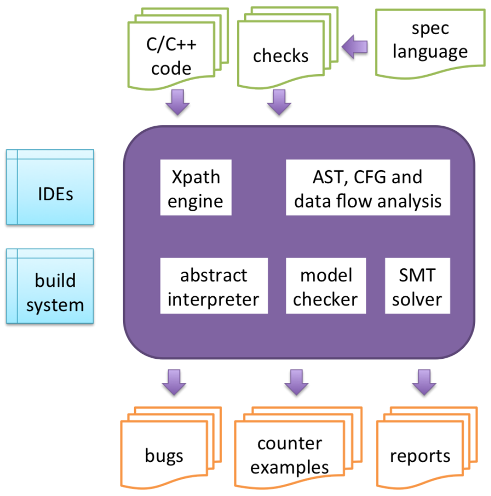

Figure 1: High-level Goanna Architecture

Business Value Interestingly, software analysis plays only a small role in the overall software development lifecycle and general product development process. Most importantly, organizations need to get products to market as fast as possible with as many features as possible. Bugs are of concern if they lead to critical failure and therefore a loss in business. A bug that never or very rarely manifests is often not of primary concern. However, of importance is being able to track the overall software quality and observe trending, see levels of compliance to standards and coding guidelines, and to obtain reports that can be understood by management. Most importantly, it is worth to note that the people making purchasing decisions are typically different from the software developers and motivated by quite different reasons.

Figure 1 depicts the high-level architecture of our software analysis tool. At the core are the different analysis engines for abstract interpretation, model checking, SMT solving and pattern matching. As input the tool takes C/C++ projects and a set of specifications that are written in our own domain-specific language and pre-defined for standard users. Goanna can be run on the command line, integrates with IDEs and can be embedded in the build system. It outputs warning messages classified by severity, displays error traces based on counter-examples, and can create summary reports.

3 Lessons

While it is a difficult endeavor to target industry with formal methods there are a range of valuable lessons that can be learned. This starts out with understanding the problem domain, the actual demands and challenges faced by the end-user as well as understanding their processes and potential *value add* by tools. Technology is only one piece of a larger puzzle.

Moreover, software developers face time constraints and pressure to deliver results. Learning a new tool from scratch is not only a process change management shies away from, but typically comes with some kind of formal training in many organization often being much more costly than the actual software tool. Hence, any tool that integrates into existing IDEs and processes that work “with the click of a button” gains much easier acceptance.

From a formal methods researcher’s point of view demands are often sobering. Real life applications require to deal with huge amounts of code in relatively short time. This will require not only abstractions, but all various heuristics, both on an algorithmic level as well as on an engineering level predicting *typical* developer behavior. Generally, the engineering requirements can easily become a roadblock for the acceptance of more advanced formal analysis techniques.

Finally, with much less effort we believe that many existing academic tools could see a larger industry uptake by following sometimes simple rules. This includes: providing a clear and end-user friendly license agreement, providing a well documented introduction to a tool, as well as a delivering degree of reliability that the software works in *most* cases. This is, however, easier said than done as academic environments typically do no provide for dedicated software engineers to develop and maintain tools over longer period of time.

References

[1] Motor Industry Software Reliability Association et al. (1998): *Guidelines for the use of the C language in vehicle based software*.

[2] Edmund M. Clarke & E. Allen Emerson (1982): *Design and Synthesis of synchronization skeletons for branching time temporal logic*. In: *Logics of Programs Workshop, New York, May 1981*, *LNCS* 131, Springer Verlag, pp. 52–71, doi:10.1007/BFb0025774.

[3] P. Cousot & R. Cousot (1979): *Systematic design of program analysis frameworks*. In: *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, New York, NY, San Antonio, Texas, pp. 269–282, doi:10.1145/567752.567778.

[4] Leonardo De Moura & Nikolaj Bjørner (2011): *Satisfiability modulo theories: introduction and applications*. *Communications of the ACM* 54(9), doi:10.1145/1995376.1995394.

[5] Vijay D'Silva, Daniel Kroening & Georg Weissenbacher (2008): *A Survey of Automated Techniques for Formal Software Verification*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 27(7), pp. 1165–1178, doi:10.1109/TCAD.2008.923410.

[6] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg & Felix Rauch (2007): *Model Checking Software at Compile Time*. In: *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, TASE '07, IEEE Computer Society, Washington, DC, USA, pp. 45–56, doi:10.1109/TASE.2007.34.

[7] Maximilian Junker, Ralf Huuck, Ansgar Fehnker & Alexander Knapp (2012): *SMT-based False Positive Elimination in Static Program Analysis*. In Toshiaki Aoki & Kenji Taguchi, editors: *14th International Conference on Formal Engineering Methods, Kyoto Japan*, *Lecture Notes in Computer Science* 7635, Springer Berlin Heidelberg, pp. 316–331, doi:10.1007/978-3-642-34281-3_23.

[8] Jean-Pierre Queille & Joseph Sifakis (1982): *Specification and verification of concurrent systems in CESAR*. In: *Proc. Intl. Symposium on Programming, April 6–8*, Springer Verlag, pp. 337–350, doi:10.1007/3-540-11494-7_22.

[9] Robert C. Seacord (2008): *The CERT C Secure Coding Standard*, 1st edition. Addison-Wesley Professional.