



Architecture of Embedded System Software

Dongdong Wang

School of Computer Technology and Automation

Tianjin Polytechnic University

Tianjin 300160, China

E-mail: forwd@126.com

Abstract

The verification of real-life C/C++ code is inherently hard. Not only are there numerous challenging language constructs, but the precise semantics is often elusive or at best vague. This is even more true for systems software where non-ANSI compliant constructs are used, hardware is addressed directly and assembly code is embedded. In this work we present a lightweight solution to detect software bugs in C/C++ code. Our approach performs static checking on C/C++ code by means of model checking. While it cannot guarantee full functional correctness, it can be a valuable tool to increase the reliability and trustworthiness of real-life system code. This paper explains the general concepts of our approach, discusses its implementation in our C/C++ checking tool *Goanna*, and presents some performance results on large software packages.

Keywords: Real-life, C/C++, Non-ANSI, Hardware

1. Introduction

Showing the full functional correctness of system software written, e.g., in C/C++, is a major challenge. It requires a precise understanding of the underlying semantics, typically needs to include an abstract hardware model, and has to give a full functional proof. There are a number of projects currently undertaking this task supported by interactive theorem provers. While this is the only way to guarantee the full correctness of a program, it requires substantial resources both in time as well as in the number of highly qualified people.

On the other hand, commercial system software has a high pressure to market, needs to run on various platforms and is rewritten frequently, making the above approach even more challenging. There are a number of lightweight analysis approaches that seek to complement full verification by detecting software bugs at the coding stage and, thus, increasing the reliability and trustworthiness of the code. Those tools make a limited but practical contribution to program correctness and can support full verification by reducing property violations in early stages.

The model-checking community has made significant advances in recent years to cover realistic C/C++ programs and produced a number of powerful tools. However, they are not yet well-suited for real-life embedded system code. On the other hand, commercial static analysis tools cope well with most C/C++ code and make a valuable contribution to software correctness. In contrast to model checking tools, static analysers typically do not allow for any user-defined specifications, but rather implement a set of independent analysis heuristics or allow specification which are less expressive than the temporal logics used by model checkers.

In this work we present a static analysis approach based on model checking. While we retain the flexibility and power of temporal logics specifications, we are able to handle any parsed C/C++ code in a uniform manner. In particular, we present the underlying idea of translating C/C++ checks into model checking properties, which can then be checked by one single analyzer, instead of a set of static analysis heuristics. In our case we use the NuSMV model checker as back end. Moreover, we present some implementation details of our checker *Goanna* and its performance on the source of large, real-life open-source software packages.

Section 2 describes our underlying framework, while Section 3 presents some of our performance results and Section 4 discusses the current state of our research as well as ongoing and future work.

2. Static Analysis by Means of Model Checking

In this section we describe how to statically check properties of C/C++ source code by means of model checking. This approach has been inspired by and is also followed by.

Using a model checker for solving static analysis problems has a number of advantages. All properties can be expressed

in a single, flexible analysis engine. This means that it is easy to add new checks by adding new checking properties. In addition, the analysis scales well with increasing number of properties. The details of our path-sensitive, intra-procedural analysis can be found in.

The basic idea is to annotate the control flow graph (CFG) of a program with atomic propositions of interest. In order to check, e.g., for uninitialized variables, we can identify atomic propositions `declq`, `readq` and `writeq`, representing program locations where a variable q is declared but not initialised, where it is read from or written to, respectively, and mark those locations in the CFG accordingly. The atomic propositions are identified by purely syntactic criteria on the abstract syntax tree (AST) of the program by means of a pattern language. We define patterns for each proposition, e.g., a variable is written to if it occurs on the left hand side of an assignment statement and so on. Once identified, the proposition is placed on the node in the CFG most closely corresponding to the nodes in the AST where it was identified.

This means we require that on all program paths if a variable q is declared it must not be read until it has been written or it will not be written at all. We use the *weak until* operator W here to include the second possibility. The latter can also point to unused variables, which is checked separately.

In the same style we can express other properties on correct pointer handling, variable usage or memory allocation and deallocation. Moreover, it allows specifying application specific properties to handle general programming guidelines, API-specific rules or even hardware/software interface rules for device drivers.

Once the patterns relevant for matching atomic propositions have been defined and the CFG has been annotated, it is straightforward to translate the annotated graph automatically into the input format of a model checker. Adding new checks only requires one to define the property to be checked and the patterns representing atomic propositions. All other steps can be fully automated.

Although this framework was developed in first instance for C/C++ it can be also extended to deal with embedded assembly code. This is important for the embedded systems space, since interaction with the hardware is frequently implemented as embedded assembly code. In particular, we take C/C++ and ARMv6 assembly interface information for our analysis into account, check for compliance of embedded assembly code with its C/C++ interface, and perform various checks on the pure assembly level. The combined analysis of C/C++ code with embedded assembly code enhances, in addition, the precision of the analysis.

3. Implementation and Evaluation

The aforementioned approach has been implemented in our program analyzer Goanna, using the open source model checker NuSMV [14] as a generic backend analysis engine. The surrounding code for pattern matching structures of interest, property definitions, CFG generation, translation into NuSMV, and representation of analysis results is written in OCaml. Moreover, Goanna can be invoked just like the `gcc/g++` compiler and, therefore, integrates seamlessly into standard development environments such as Eclipse.

We evaluated Goanna on a number of open source packages ranging from highly optimized system software such as the L4 microkernel1 to large application code bases such as the 260 kLoC2 Open SSL package. For an unoptimized version of Goanna some run-time results for Open SSL are shown in Figure 3. It shows that over 80% of all files are analyzed within 1 second and that 99% of all files are analyzed within 5 seconds. The whole analysis takes less than 15 minutes. Proportionally, the time spent purely in NuSMV is mostly negligible with 98.7% of all files being analyzed in less than 2 seconds.

The run-times of Figure 3 are based on checking for 15 properties ranging from simple uninitialized variables, over potential null-pointer dereferences, to memory leaks. It is worth to mention that increasing the number of properties typically scales well in our framework as it only increases the number of labels and property specifications in the same NuSMV model, which is handled well by the model checker. For instance, increasing the number of properties from one to 15 only doubled the overall analysis time.

Moreover, we found that the analysis time is not only well within the same order of magnitude as the compile time, but that the memory requirements of the analysis fit easily in the RAM of current developer machines.

The analysis of C/C++ code with embedded assembly code was evaluated for Pistachio 0.4 implementation3 of L4, compiling for an ARM SA1100 architecture. It contains 54 C++ files, two of which have embedded assembly blocks (3.7%), and they include a total of 72 header files, of which 10 have embedded assembly blocks (13.8%). The additional assembly analysis lead to a modest increase from 75.9 seconds to 77.3 seconds, which is an increase of only 1.4 seconds or 1.8%.

4. Conclusion

In this work we presented our framework and results on model checking system software by means of static analysis. We showed how to easily encode static checks as model checking properties, providing the basis for an extendable and

flexible checker. Moreover, we implemented our analysis framework in Goanna, the first static checker using NuSMV as its analysis engine, and presented some run-time and scalability results. We showed that this is a viable solution that can be integrated well in the software development process.

References

Karim. (2003). *Building Embedded Linux System*. USA:O'Reilly.

P. Raghavan. *Embedded System Design and Development*. Auerbach Publications.

R. Allen. A Formal Basis for Architectural Connection (1997). *ACM Transactions on Software Engineering and Methodology*, pp. 213-249.