Automatic Heavy-weight Static Analysis Tools for Finding Bugs in Safety-critical Embedded C/C++ Code

David Farago, Florian Merz, Carsten Sinz (<forename>.<surname>@kit.edu)
Karlsruhe Institute of Technology, Institute for Theoretical Computer Science

May 7, 2014

Abstract

This paper motivates the use of automatic heavy-weight static analysis tools to find bugs in C (and C++) code for safety-critical embedded systems. By heavy-weight we mean tools that employ powerful analysis to cover all cases. The paper introduces two automatic and relatively heavy-weight tools that are currently employed in the automotive industry, and depicts their underlying techniques, advantages, and disadvantages. Since their results are often imprecise (false positives or false negatives), we advocate the use of alternative techniques such as software bounded model checking (SBMC), which can achieve bit-precise results. Finally, the tool LLBMC is described as an example of a tool implementing SBMC, which makes use of satisfiability modulo theories (SMT) decision procedures as well as the LLVM compiler framework.

1 Introduction

The number of *embedded* (or cyber-physical) systems increases permanently, and they take over more and more *safety-critical* tasks. Correctness of these systems is often an issue, and difficulties in quality assurance are still prevalent and not yet solved [23]. This is also the case for embedded systems in the automotive industry: while the number of recalls per year decreases, the rate of vehicles affected by software failures increases, so that in 2013 nearly every second recall was due to software related issues [17, 1]. This is not astonishing, since about 90% of all automotive innovations are based on electronics and software [1].

Since most embedded code is written in an imperative programming language, mainly C [1], its most frequent *fault classes* are:

- *memory faults*, like illegal access (array index out of bound or buffer overflow, illegal pointer access or null pointer dereference), memory leaks, invalid or double free, use after free;
- *arithmetic faults*, like integer under-/overflow and division by zero;
- *bit operation faults*, like invalid bit shifts.

Automotive embedded C code is often developed in accordance to standards such as ISO-26262 [15, 7] and MISRA-C [8]. Especially the latter standard contains an extensive list of requirements that restrict the allowed C language features, thereby trying to avoid the most common pitfalls encountered when programming in C. MISRA-C covers a wide range of topics. The most notable rules are concerned with disallowing dynamic memory management and recursion, and restricting loops and the use of loop counter variables with the goal of reducing the risk of infinite loops.

To automatically find these kinds of faults, industry employs automatic static analysis tools, as described in the next section.

2 State of the Art

2.1 Approaches and Tools

Many software quality assurance tools are based on light-weight, heuristical approaches, such as pattern matching. Examples are Secure Programming Lint (Splint) [21, 9, 6], Cppcheck [13, 9], or QA·C [19]. Due to the heuristics employed, the tools may be not precise enough for safety-critical code: in many situations, they cannot detect whether a fault occurs or not. This either leads to many false warnings (false positives) or to missed faults (false negatives); the term *incorrect result* summarizes both. Therefore, this paper focuses on automatic heavy-weight approaches and tools for safety-critical embedded C/C++ code, which try to avoid these shortcomings.

Many heavy-weight tools use approaches that check the conformance of the code to some formal specification, e.g. using design by contract. Examples are VCC [22], BLAST [11] and SLAM [20]. Each of these tools offers its own specification language. Such formal specifications can be more expressive and concise than user assertions formulated in C. (They might, e.g., allow expressions using quantifiers.) Using a different language for specifications can also avoid making the same mistake in the production code and in the specified property. But as additional formal specifications add costs, this paper does not cover those approaches and tools.

Unfortunately, analyzing source code in a heavy-weight manner that covers all cases can quickly become too complex due to the amount of possible ex-

ecution paths. The tools described in this section, Polyspace and Coverity, use *abstract interpretation* [4] over the whole code base as technique to achieve decidability and scalability: the *concrete semantics*, represented by the set P of all possible execution paths of the program, are abstracted, leading to an approximate program with a superset of execution paths $P_{abstract}$.

The abstraction is achieved by partitioning variable domains. For instance, the domain of a signed integer variable v by only storing its sign, i.e. reducing v's domain from 2^{32} values to the three values $\{-, 0, +\}$. Therefore, 2^{31} states with negative values for v are represented by one abstract state with the value "$-$" for v. Due to the lost details, $P_{abstract}$ might contain some paths outside of P, possibly causing false positives. Therefore, abstract interpretation also has to cope with imprecision, but not as much as light-weight approaches. The applied abstractions depend on both the properties and the program being checked. Due to the abstractions, all tools based on abstract interpretation cannot give detailed and concrete information about the detected faults, making it very difficult to understand warnings and to figure out whether they are false positives.

Polyspace [18, 6, 5] is one of the first industrial heavy-weight tools for checking C/C++ code, as well as Ada. It is extensively applied for embedded software. Since it uses abstract interpretation, it tries to cope with imprecision by differentiating warnings that it knows are true (marked green) and those which might be false positives (marked orange). Since the amount of orange warnings often becomes large, their manual post-processing becomes expensive. Therefore, orange warnings are often still considered as false positives, although orange gives a hint; the false positive rate can be as high as 50% [6, 5], i.e. half the warnings are orange. Fortunately, green warnings are never false positives. The rate of false positives per line of code is slightly above 2%.

Coverity Code Advisor [12, 6, 5] started 10 years ago and has become a popular automatic heavy-weight tool for checking C/C++ code, as well as Java and C#. It uses abstract interpretation, mainly with interval ranges and simple relationships between local variables as abstraction mechanism, resulting in weak performance for global variables and aliasing. Coverity copes with the high amount of false positives of abstract interpretation by trying to detect which warnings might be false positives and suppressing them (instead of marking them orange as Polyspace does). Due to the employed abstractions, Coverity cannot detect this precisely, leading to many false negatives. In summary, Coverity has a smaller false positive rate of about 15%, but also contains false negatives.

There are further heavy-weight tools for checking C/C++ code, for instance Astrée [10], Klocwork [16, 6, 5] and Frama-C [14]. This paper does not introduce them because they also use abstract interpretation and hence behave similarly. Furthermore, they are hardly ever used in the automotive industry [1].

2.2 Advantages and Disadvantages

To investigate the advantages and disadvantages of the static analysis tools, we use the following criteria:

- How many incorrect results occur?
- Which fault classes are covered, and how precisely?
- Are user assertions supported?
- Can nontermination be detected?
- Are multiple languages supported?
- Can analysis be performed incrementally?
- How well/much information is provided for a detected fault?
- How laborious is the setup and configuration for a static analysis run?

The last four items are very important since they determine the usability of the tool within the software development life-cycle.

Polyspace yields many incorrect results, as described in the previous subsection. It covers all fault classes mentioned above, but does not detect all faults in each class. It can check user assertions and nontermination. The setup for a Polyspace run is large since all code, project and build files, as well as abstract interpretation heuristics must be configured. Furthermore, it does not re-use information from previous runs, i.e. each run of Polyspace is independent, causing a lot of overhead in the software development life-cycle. Since Polyspace has been bought out by MathWorks, it integrates very well with Simulink, supporting model-based design and traceability.

Coverity also yields many incorrect results, partly false negatives and partly false positives. It covers all fault classes mentioned above, but does not detect all faults in each class. It can check user assertions to some extent, but not nontermination. No changes to the code or build system are required, making setup and configuration very fast. Furthermore, it supports iterative analysis using impact analysis. Therefore, Coverity can efficiently be integrated into the software development life-cycle.

A summary is given in Table 1.

3 LLBMC

3.1 Approach

LLBMC is a software bounded model checking tool developed at the Karlsruhe Institute of Technology since 2009.

Bounded model checking was first presented in [2] for the formal verification of hardware systems. It was designed to taking advantage of recent performance improvements in SAT solvers. Since then, bounded model checking has become a well established method in hardware verification.

CBMC [3, 9] was the first tool to successfully apply bounded model checking to real life software systems, by translating a software system to a propositional formula and using SAT solvers for determining the formula's unsatisfiability and thereby the system's correctness.

LLBMC, while following CBMC's general approach, employs an SMT solver instead of a SAT solver. This added layer of abstraction enables further performance improvements.

Currently, LLBMC focuses on supporting C/C++, but because LLBMC is based on the LLVM compiler framework, LLBMC will automatically support further languages as soon as LLVM supports them.

In software bounded model checking, as implemented in LLBMC, first all loops in the program are unrolled and all function calls are inlined. Note, that this is especially well-suited for programs following the MISRA-C [8] rules. The resulting non-looping, single-function program is then translated to an intermediate logic representation, which uses static single assignment form and an explicit representation of the memory state. The resulting representation of the program is entirely stateless and can be translated into an SMT formula straightforwardly. With the help of an SMT solver, this formula is then checked for satisfiability. If the formula is satisfiable, the model created by the solver can be translated into a counter example for the software system. If the formula is unsatisfiable the checked properties are not violated by the program.

3.2 Advantages and Disadvantages

This approach means LLBMC is well suited for programs, for which loop and recursion bounds are fixed and ideally, though not strictly necessarily, known in advance.

Due to LLBMC's use of the theory of bitvectors instead of mathematical integers the analysis is bit-precise. This means for example the effects of arithmetic overflows are always handled correctly, as are any of the usual bitwise operations.

Because LLBMC does not rely on finding a suited abstract domain, it can more easily support arbitrary user assertions. If a property can be expressed in C (or C++) it can be checked by LLBMC and preliminary support for more expressive specifications is being worked on.

LLVM's C and C++ frontend clang is already GCC compatible and support for MSVC compatibility is actively being developed. This not only covers aspects like language extensions provided by the different compilers, but also command line compatibility. LLBMC greatly benefits from this extensive built-in tooling support.

What is more, LLBMC's choice of LLVM as a frontend greatly reduces the efforts required for supporting languages other than C/C++, and, if need arises, LLVM's GCC plugin dragonegg even allows verification of those languages supported by the GNU compiler collection such as Ada or Fortran.

Finally, one of the most powerful features of LLBMC is its ability to generate complete, bit-precise counter examples for any fault it uncovers. These counter examples provide all necessary information to localize the cause for the fault.

LLBMC proved itself already in the anual SV-COMP software competition, winning four gold, six silver, and four bronze medals during the last three years. Furthemore, encouraged by a successful multi-year collaboration with Daimler AG, founding of a start-up company is currently planed and already funded by the Helmholtz Association.

4 Conclusion

This paper has motivated automatic heavy-weight static analysis without formal specifications, especially for safety-critical embedded systems. For decidability and scalability, any approach must approximate the concrete semantics of the code.

The industrial tools Coverity Code Advisor and Polyspace use abstract interpretation, which leads to higher precision than light-weight tools, but still many incorrect results. The tool LLBMC uses software bounded model-checking, which is a suitable alternative, especially for embedded systems. Thus LLBMC better satisfies most criteria, as summarized in Table 1.

References

[1] H. Altinger, F. Wotawa, and M. Schurius. Testing methods used in the automotive industry – results from a survey. In *JAMAICA 2014*, 2014. Submitted.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC'99*, pages 317–320, New York, NY, USA, 1999. ACM.

[3] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS 2008*, volume 2988 of *LNCS*, pages 168–176, 2004.

[4] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics 2001 (Dagstuhl 10th Anniversary)*, volume 2000 of *LNCS*, pages 138–156. Springer, 2001.

[5] P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electr. Notes Theor. Comput. Sci.*, 217:5–21, 2008.

[6] I. Gomes, P. Morgado, T. Gomes, and R. Moreira. An overview on the static code analysis approach in software development. Technical report, Faculdade de Engenharia da Universidade do Porto, 2009.

[7] M. Hillenbrand. *Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen*. PhD thesis, KIT, 2011.

[8] MISRA C: Guidelines for the Use of the C Language in Critical Systems 2012.

[9] L. Torri, G. Fachini, L. Steinfeld, V. Camara, L. Carro, and E. Cota. An evaluation of free/open source static analysis tools applied to embedded software. In *11th LATW*, pages 1–6. IEEE, 2010.

[10] The Atrée Static Analyzer. `http://www.astree.ens.fr/`. Accessed: May 2014.

[11] MTC: BLAST project. `http://mtc.epfl.ch/software-tools/blast/index-epfl.php`. Accessed: May 2014.

[12] Software Development Testing and Static Analysis Tools. `http://www.coverity.com/`. Accessed: May 2014.

[13] CppCheck - A tool for static C/C++ code analysis. `http://cppcheck.sourceforge.net/`. Accessed: May 2014.

[14] Frama-C home page. `http://frama-c.com/`. Accessed: May 2014.

[15] ISO Online Browsing Platform - ISO 26262-1:2011(en). `https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en`, 2014. Accessed: May 2014.

[16] Source Code Analysis Tools for Software Security & Reliability - Klocwork. `http://www.klocwork.com/`. Accessed: May 2014.

[17] Recalls database of the National Highway Traffic Safety Administration. `http://www-odi.nhtsa.dot.gov/owners/RecentInvestigations`, 2014. Accessed: May 2014.

[18] Static Analysis Tools for C/C++ and Ada - Polyspace. `http://www.mathworks.com/products/polyspace/`. Accessed: May 2014.

[19] QAC and QAC++ Static Analysers. `http://www.phaedsys.com/principals/programmingresearch/index.html`. Accessed: May 2014.

[20] SLAM - Microsoft Research. `http://research.microsoft.com/en-us/projects/slam/`. Accessed: May 2014.

[21] Splint home page. `http://www.splint.org/`. Accessed: May 2014.

[22] VCC: A C Verifier - Microsoft Research. `http://research.microsoft.com/en-us/projects/vcc/`. Accessed: May 2014.

[23] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

Table 1: Which criteria are fulfilled how well by each tool?

Criterion	Tool		
	Coverity	PolySpace	LLBMC
Detection of...			
memory faults	some	some	all
arithmetic faults	all	all	all
bit operation faults	some	some	all
Support for...			
user assertions	weak	strong	strong
nontermination	no	yes	yes (within bounds)
multiple languages	C/C++/Java/C#	C/C++/Ada	C/C++ (and all LLVM)
incremental analysis	yes	no	no
Miscellanea			
fault information	limited	limited	full counterexamples
setup effort	little	medium	little (via LLVM)
incorrect results	many	many	few or none