Static Program Analysis for Real-Time and Embedded Systems

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University
Västerås, Sweden
bjorn.lisper@mdh.se

Abstract

Static program analysis methods can find properties of software without running it, by analyzing a mathematical model of the software. The analysis can be designed to detect potential bugs, and thus provides an interesting alternative to testing. Static analyses can also estimate quantitative properties like execution time, and memory consumption. Contrary to testing, static analysis provides formal evidence whether a property holds: thus, its results can be trusted with a high degree of confidence. This makes the technique very interesting to use in the development of embedded systems, where the demands on functionality, stability and safety are high.

The Programming Languages group at Mälardalen University has been active in the static program analysis area since more than ten years. The main focus has been on Worst-Case Execution Time (WCET) analysis, *which finds safe upper bounds to the execution time of a program, and the group is one of the world-leading groups in this area. However, the techniques and tools developed by the group have a number of other potential applications as well for embedded systems development. Here we give an introduction to static analysis, we describe our techniques and our current research, and we hint at some possible applications.*

1 Introduction

Throughout the process of software development, there is a need to ensure that the developed system meets its requirements. Traditionally, this is done through extensive testing. However, testing is time-consuming and error-prone: since it typically is impossible to perform exhaustive testing, there is always a risk that some bug goes unnoticed.

Testing is done by running the software. In contrast, *static program analysis* [11] finds out about properties of the program without running it, analyzing a mathematical model. This is analogous to traditional engineering disciplines, where constructions are analyzed using some numerical method before deployed.

Static program analysis can be performed both on source- and machine code. Source-level analysis is usually easier, since there is more information and structure in source code than in machine code. However, some properties are pertinent to machine code and thus cannot be analyzed faithfully on source level: an example is execution time. Furthermore, small embedded systems are often programmed in assembly code or a mix of high-level code and assembly: for them, a pure source-level analysis will not be applicable.

There is a range of static program analysis techniques. Simple syntax checking can find certain kinds of suspicious code constructs: the classical tool "lint" uses this kind of analysis to check C programs [9]. More advanced techniques are *semantics-based*: obviously, such techniques can be much more general and powerful since the interesting properties of software typically are more related to its semantics than its syntax.

Semantics-based static analysis first appeared in optimizing compilers. Such compilers employ various *data flow analyses* that help decide when certain code optimizations can be performed without breaking the program. Data flow analyses can also be used to aid program verification: an example is to check for possible uses of uninitialized variable values.

Another class of semantics-based static analyses is based on *abstract interpretation* [4]. Abstract interpretation works by connecting a *concrete domain*, where the semantics of the program is described, to a simplified *abstract domain* where the analysis is performed. An important example is *value analysis* of imperative programs, where the possible values for different program variables are approximated in different program points. The results of a value analysis can be used for a variety of verification purposes, like checking for potential out-of-bounds array accesses, or possible divisions by zero.

For real-time systems, methods for *Worst-Case Execution Time (WCET) analysis* have been developed [12]. The WCET is the longest possible execution time for a program executing uninterrupted on a certain hardware: good estimates are needed when analyzing hard real-time systems with respect to deadline violations. WCET analysis attempts to find a safe (not underestimating) WCET esti-

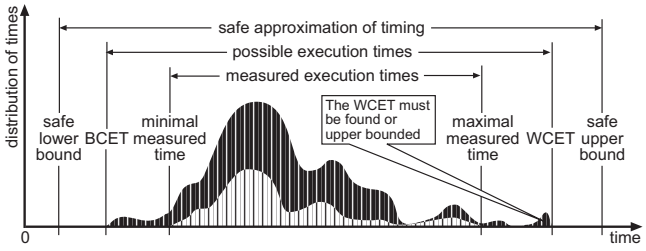

Figure 1. WCET, BCET (Best-Case Execution Time), safe bounds, and possible measured times.

mate from information about possible program flows and hardware timing models: see Fig. 1. WCET analysis is typically done on the linked binary, since this is the code actually executing on the hardware.

Static program analyses decide whether a program has a property P. They are usually designed to be *sound*: if they say that a program has the property P then this is surely true. On the other hand, they are usually not *complete*: the analysis might say that the program possibly does not have the property P, and then there is a possibility that it indeed fulfils P even though the analysis couldn't say so. If P is the absence of some bug, then this situation is called a *false positive*: the tool reports a potential bug that in fact is not present. Obviously, the number of false positives must be low for the analysis to be useful.

The soundness of static program analysis makes it dual to testing. If testing finds a bug, then it is clearly a true bug and not a false positive. On the other hand, testing can never be sound (unless exhaustive): we can never prove the absence of a bug for sure by testing alone. Thus static analysis and testing are complementary techniques, and it is very interesting to use them in combination.

2 Tools

For a long time, semantics-based static program analysis was considered a purely academic discipline of little practical interest. However, during the last decade commercial tools have appeared, and they are now being increasingly used by software developers. Coverity[1] uses a variety of static analysis techniques, originating in optimizing compilers, to perform software verification. Polyspace[2] and Astrée[3] both perform value analyses based on abstract interpretation in order to prove the absence of run-time errors. Klocwork Insight[4] uses data flow analysis and symbolic logic. CodeSonar[5] is based on a combination of data flow analysis and symbolic execution.

These tools work for languages like C, C++, C#/Java. They can typically detect run-time errors like division by zero, null pointer dereferencing, uninitialized variables, out-of-range indexing, and a number of other errors.

In the area of WCET analysis, commercial tools are provided by AbsInt[6], Tidorum[7], and Rapita Systems[8]. The two first are based on pure static analysis of linked binaries, whereas the tool from Rapita Systems uses a hybrid approach combining timing measurements with static analysis techniques.

3 Static Analysis Research at MDH

The Programming Languages group[9] is part of Mälardalen Real-Time Research Centre (MRTC)[10]. The group is internationally established in the area of WCET analysis[11]. A speciality of the group is *program flow analysis*, which derives constraints on possible program flows. Such constraints can be upper bounds on the number of loop iterations, or constraints expressing mutual exclusion between different program parts. We have developed several analyses for this purpose [5, 7, 10].

We use our prototype WCET analysis tool SWEET (SWEdish Execution Time tool)[12] to evaluate our methods. The tool performs program flow analysis on the intermediate format ALF [6], which is designed to be able to represent both C code and object code faithfully. SWEET can analyze other formats than ALF through frontends mapping these formats to ALF. Currently, frontends for C and PowerPC binaries exist. In addition to program flow analysis, SWEET is also capable of other analyses such as an advanced value analysis based on abstract interpretation, and program slicing. All these analyses are interesting in themselves: thus, SWEET is now developing into a tool for general static analysis of embedded code.

We are currently working on a number of research topics:

- *Approximate source-level WCET analysis* [1]. Conventional WCET analysis requires linked binaries, and can thus be applied only late in the development process. It is often desirable to estimate the timing properties earlier in the process, to reduce the risk for a costly system redesign if the timing requirements turn out not to be fulfilled in the end. We have developed a method to derive approximate source-level timing models which can be used to perform a rough, static WCET analysis for source code targeting a certain hardware.
- *Parametric WCET analysis* [2, 10]. Conventional WCET analysis returns a single number. However, many time-critical tasks actually have a parametric timing behaviour where the execution time depends

[1] www.coverity.com
[2] www.polyspace.com
[3] www.absint.de/astree/
[4] www.klocwork.com
[5] www.grammatech.com
[6] www.absint.de
[7] www.tidorum.fi
[8] www.rapitasystems.com
[9] www.mrtc.mdh.se/index.php?choice=research_groups&id=0009
[10] www.mrtc.mdh.se
[11] www.mrtc.mdh.se/projects/wcet/
[12] www.mrtc.mdh.se/projects/wcet/sweet.html

strongly on some input values. For such tasks, a single number provides a very crude overapproximation. We are developing techniques to perform a *parametric* WCET analysis, where the answer is a formula in the input values rather than a single number. Our techniques use a number of advanced symbolic analysis methods.

- *Bit-precise value analysis* [3]. For small embedded processors (8/16 bit), the finite wordlength of the arithmetics is of significance. Unintentional overflows may occur, as well as intentional use of overflow with value wraparounds to save instructions. As far as we know, no current commercial tool takes such effects into account: thus, their analyses are unsound for such software and their results cannot be trusted. We have developed an advanced value analysis that takes possible overflows into account, and implemented it in SWEET. This analysis is sound also in the presence of wraparounds.
- *WCET Analysis of parallel software* [8]. With the rapid introduction of multicore processors, analysis of parallel software is becoming a very pressing issue. We are investigating methods to perform WCET analysis for parallel software. This is a very hard problem, and the analysis of hard real-time systems on multicore processors probably requires a very stringent system design to be feasible.

4 Potential Topics for Collaboration

We are very interested to get in touch with companies that have an interest in static analysis techniques, to discuss possible future collaboration. We believe that we have a clear edge as regards precise and sound methods for static analysis of small embedded systems. We are also interested to further our techniques how to build models for performance prediction. Here is a non-exhaustive list of potential topics of collaboration:

- Sound and precise static analysis of embedded software
- WCET analysis of real-time systems
- Systematic methods to build models for performance prediction
- Combining static analysis and testing
- Static analysis for multicore

5 Conclusions

Semantics-based static program analysis is becoming an important tool for software developers. The trustworthiness of the analysis makes it very interesting for verification of embedded software, where demands on correct functionality and stability are high. The Programming Languages group at Mälardalen University performs research in the area, and is interested in future research collaborations with companies that have an interest in these techniques.

References

[1] P. Altenbernd, A. Ermedahl, B. Lisper, and J. Gustafsson. Automatic generation of timing models for timing analysis of high-level code. In S. Faucou, editor, *Proc. 19th International Conference on Real-Time and Network Systems (RTNS2011)*, Nantes, France, Sept. 2011.

[2] S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric WCET calculation. *Journal of Systems Architecture*, 57:614–624, 2011.

[3] S. Bygde, B. Lisper, and N. Holsti. Fully bounded polyhedral analysis of integers with wrapping. In *Proc. Int. Workshop on Numerical and Symbolic Abstract Domains (NSAD 2011)*, Venice, Italy, Sept. 2011.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4^{th} ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, Jan. 1977.

[5] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In C. Rochange, editor, *Proc. 7^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2007)*, Pisa, Italy, July 2007.

[6] J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, and L. Källberg. ALF – a language for WCET flow analysis. In N. Holsti, editor, *Proc. 9^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2009)*, pages 1–11, Dublin, Ireland, June 2009. OCG.

[7] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27^{th} IEEE Real-Time Systems Symposium (RTSS'06)*, Dec. 2006.

[8] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In B. Lisper, editor, *Proc. 10^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 103–113, Brussels, Belgium, July 2010. OCG.

[9] S. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Dec. 1977.

[10] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In J. Gustafsson, editor, *Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, pages 77–80, Porto, July 2003.

[11] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2^{nd} edition*. Springer, 2005. ISBN 3-540-65410-0.

[12] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.