Object Code-Near Software Verification

Jan Tobias Mühlberg

University of Applied Sciences in Brandenburg
14776 Brandenburg/Havel, Germany
`muehlber@fh-brandenburg.de`

27th May 2005

Contents

1 Introduction **1**

2 Correct Software? **1**
2.1 Testing 2
2.2 Static analysis 2
2.3 Model checking 2
2.4 Semantic verification 2

3 Reasoning on formal proofs **2**

4 A new approach in software verification **3**

5 Main research questions **3**

References **4**

1 Introduction

Computer programs are used in almost every area of our daily life. Especially in fields like medical computing and the whole sector of security systems they became almost vital or at least mission critical. One of the main problems of software engineering is to develop programs that are reliable enough to meet the needs of a mission critical appliance. Several approaches are taken to prove that software is correct, e.g. to show that it does what it should do and thereby establish trust in the software. This work gives an outline on the idea of formal software verification using the intermediate representation language RTL of the GNU Compiler Collection.

2 Correct Software?

Programs are developed to cause a certain kind of behaviour from a computer. This behaviour is usually described within the specification of the program. If a program satisfies its specification it is called correct: every behaviour produced by the program has the properties described within the specification. The correctness problem is to prove that a given program meets its specification. The mainly used ways of drawing close to correct software are by testing the program, performing static analysis, model checking and by semantic verification.

2.1 Testing

Testing a program means to provide some input to it and to check whether it produces an output or behavior as expected. Applying this method is only useful to find errors in the software. It can not be used to prove that the program is correct. Furthermore it will only show that the program produces the expected behaviour for the test vectors but for all possible input. Methods for software testing are extensively discussed in literature on software engineering.

2.2 Static analysis

Static analysis is usually run on the whole source code of a software and will find possible programming errors like overflowable buffers or problems with type conversions. Likewise software testing, it will not prove the correctness of a program but may increase trust in it. There are several tools for for static program analysis available, one of the more well-known is the Meta-Level Compilation project [1].

2.3 Model checking

Model checking means to develop an abstract model of a program and to prove that this model satisfies the formal specification of the program. While the construction of the model needs to be done by hand[1], the verification is done by a model checker like SPIN [8, 7] by comparing possible states of the system. If the model checker fails to prove a specified property of the model an error is found.

Engler [2] gives an overview on the advantages and disadvantages of static analysis and model checking. He believes that "static analysis will greatly win in terms of finding as many bugs as possible" and its performance but notes that model checking finally gives much stronger correctness results. Despite this he states that "no model is as good as the implementation itself. Any modification, translation, approximation done is a potential for producing false positives, danger of checking far less system behaviors, and of course missing critical errors".

2.4 Semantic verification

In order to perform a semantic verification of a program, a semantic domain for its programming language needs to be determined. This domain is a set of mathematical objects being appropriate to represent the bahaviour of all possible programs. A compiler is needed to translate the program code into terms of logic by using the semantic. Besides this, the formal specification of the program needs to be available too. Program verification is then done by a theorem prover like PVS [10]. Amongst others, there are ongoing projects dealing with the development of semantics for C, C++ [12, 6, 13] and the verification of operating systems [5, 4].

3 Reasoning on formal proofs

Both, model checking and semantic verification are methods capable of establishing a very high level of trust into the correctness of a given program. To differenciate the plausibility of both methods it is neccessary to keep a close look on what is verified: When model checking a program we are reasoning on an abstract model of the software while a semantic verification employs the whole source code of it. Thus, the semantic verification

[1]A more automatic way of using a model checker is introduced by Holzmann [9].

should be the more reliable.

Programs are typically written in high-level languages. These languages have only very few in common with the so called object code – the processor language. Because of this the high-level program needs to be compiled into object code before it can be run. As a result of this, verifying a program in high-level language actually does not prove that the resulting object code program is correct. There are approaches for object code verification [15, 14]. What makes them lacking practical usability is that there are huge differences between the object code generated for different processors. Therefore even the most platform independent high-level program needs to be verified separately for each runtime environment.

4 A new approach in software verification

A possible way of dealing with this problem could be developed using the internal representation languages of compilers such as the GNU Compiler Collection (gcc) [3]. The gcc supports the compilation of several high-level languages such as C, C++, Fortran, Ada and Java into object code for a wide variety of runtime environments. This is done by using an intermediate representation called register transfer language (RTL) [11]. In RTL an instruction is described in an algebraic form that contains information on what the instruction does. Actually much of the optimization done by the compiler is applied to the RTL representation of a program. This renders RTL an almost perfect platform for doing formal verification:

- Verification can be done independent to the high-level language the program was developed in. There will be no need to use different semantics for Java, C and others.
- Thereby problems resulting from the informal description [5, 15] of these languages can be avoided. Since RTL describes what a program does in an algebraic way it is much more suitable for verification.
- The verification can be done on a level much closer to object code than by using the high-level language.
- The results of a verification will apply for several operating environments.

Of course there are a few drawbacks arising with the use of RTL. Those include that it is done by the compiler internals and therefore lacks information processed by a preprocessor. On the other hand RTL code is getting more and more platform dependent while optimisation and other transformations are applied to the code. Therefore the RTL may not always contain all the information about the program. It could be tricky to find the right moment in the compilation process to take out the program for verification.

5 Main research questions

There are several initial questions to be answered before software verification using RTL might be practicable:

- RTL is quite a complex language but rather close to object code. Would it be possible to use a language such as $\mathcal{L}$ [15] to verify RTL programs?
- To what extend can RTL instructions be abstracted in order to simplify the verification process?
- What other optimisations could be done to make a verification applicable in classic regression tests, e.g. compile and verify a

program automatically once a day to reflect the requirements of rapid prototyping.

The most interesting point of the research would be to use these verification mechanisms to prove the correctness of highly critical components of operating systems. This could be done on the example of the cryptographic interface or the upcoming implementations of drivers for the TCPA trusted platform module for the Linux kernel.

References

[1] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, October 2000. Available online at `http://www.stanford.edu/~engler/`; visited 26th May 2005.

[2] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, January 2004. Available online at `http://www.stanford.edu/~engler/`; visited 26th May 2005.

[3] The GNU Compiler Collection. `http://gcc.gnu.org/`.

[4] Christoph Haase and Hendrik Tews. Verifiziertes Fiasco. In *Proceedings of the 21st Chaos Communication Congress*, December 2004.

[5] Michael Hohmuth, Shane G. Stephens, and Hendrik Tews. Applying source-code verification to a microkernel – The VFiasco project. Technical report, Dresden Technical University, Germany, March 2002.

[6] Michael Hohmuth and Hendrik Tews. The Semantics of C++ Data Types: Towards Verifying low-level System Components. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, September 2003.

[7] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1 – 17, May 1997.

[8] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley Longman, Amsterdam, 2003.

[9] Gerard J. Holzmann and Rajeev Joshi. Model-Driven Software Verification. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, 2004. Available online at `spinroot.com/spin/Workshops/ws04/036-Holzmann.pdf`; visited 26th May 2005.

[10] PVS Specification and Verification System. `http://pvs.csl.sri.com/`.

[11] The Register Transfer Language. `http://gcc.gnu.org/onlinedocs/gccint/RTL.html`.

[12] Hendrik Tews. Coalgebraic Methods for Object-Oriented Specification. Dissertation, Dresden Technical University, Germany, 2002.

[13] Hendrik Tews. Verifying Duff's device – A simple compositional denotational semantics for Goto and computed jumps. Technical report, Dresden Technical University, Germany, 2004.

[14] Matthew Wahab. Verification and Abstraction of Flow-Graph Programs with Pointers and Computed Jumps. Technical report, University of Warwick, UK, November 1998.

[15] Matthew Wahab. Object Code Verification. Dissertation, University of Warwick, UK, December 2002.