

Safety of Compilers and Translation Techniques - Status quo of Technology and Science

Stephan Frank, Martin Grabmüller, Petra Hofstedt, Dirk Kleeblatt, Peter Pepper
Technische Universität Berlin
{sfrank, magr, ph, klee, pepper}@cs.tu-berlin.de

Pierre R. Mai
PMSF IT Consulting
pmai@pmsf.de

Stefan-Alexander Schneider
BMW Group
stefan-alexander.schneider@bmw.de

Abstract

The paper investigates the status quo of technology and science for compiler safety methods. We consider specific requirements for compilers and generators for automotive applications and discuss established and newly emerging methods of compiler safety in this context. We put the main focus on compiler verification methods on the one hand and testing technology on the other hand and discuss their applicability in the automotive context.

1. Introduction

We investigate technologies for ensuring safety of compilers, where we do not only consider the status quo of technology but take into consideration recent developments and progress in science which presumably will play a role in the (near) future.

A compiler (or code generator) translates a program given in a high level programming language (HLL) or a visual language into machine code or code of another HLL. If one succeeds to prove the safety and correctness of a compiler, then for every application translated using this compiler, one only needs to ensure the correct modeling in the source language, but can omit an investigation of the compilation result.

Methods for ensuring compiler safety are based on a variation of ideas: While one approach is to restrict the source language, such that potential sources of problems are avoided a priori, other methods try to formally prove the correctness of certain translation steps or rely on testing. While compiler safety is a fundamental item for automotive applications because of very high reliability and correctness requirements for the basis software, there are only few, restricted results published so far. The aim of our analysis and of this paper is to investigate and classify the established techniques and newly emerging approaches in general and in particular with respect to their applicability in the automotive context.

This article is structured as follows: Section 2 gives a brief introduction to the area of compiler technology and compiler safety methods and describes the specific circumstances in modern automotive software development. Based on these observations, we derive compiler requirements in the automotive application context. In Section 3, we classify the wide spectrum of methods used to ensure compiler safety into three classes. We shortly describe their main ideas and examine published methods applied in automotive developments. In the main part of the paper, we inspect in detail the two main classes which cover most of the current approaches: We consider methods for compiler verification on the one hand in Section 4 and test-based approaches which are located on the other end of the spectrum of methods in Section 5. Section 6 summarizes our analysis.

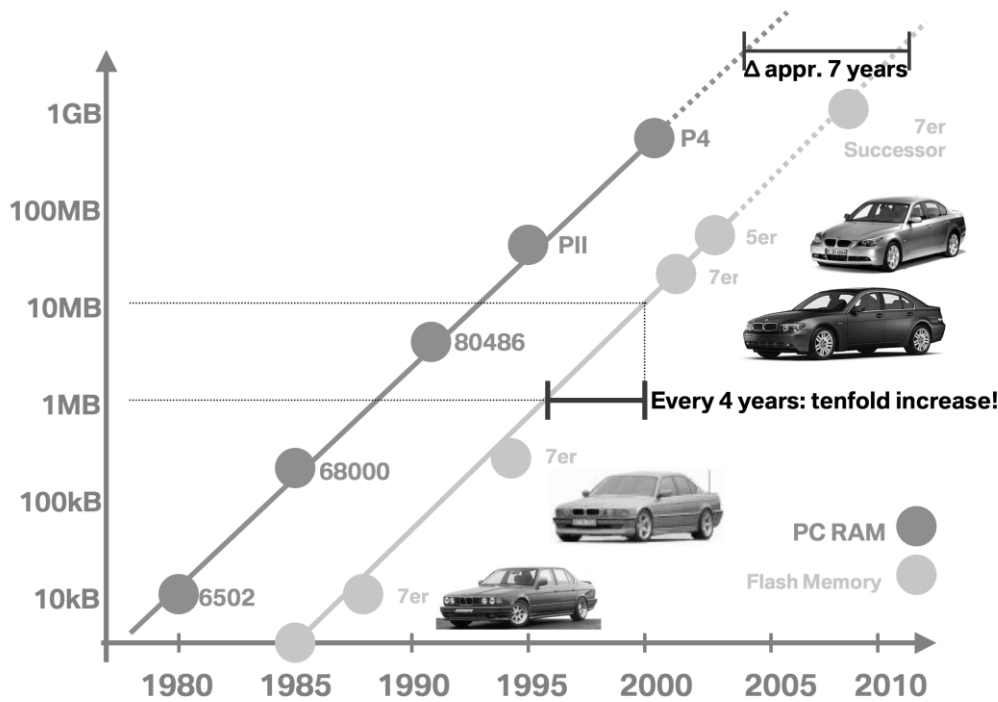


Figure 1: Moore's Law and automotive Software (Source: BMW AG, 2004)

2. Compilers and Compiler Requirements in the Automotive Context

A compiler translates a program given in an HLL like C, C++, or Java, in a visual language, like MATLAB & Simulink, Modelica, or others into either machine code or into code of another HLL. In the latter case, the compiler is often called generator. In keeping with compiler design nomenclature, we will use the notion “compiler” in the following for either case.

The compilation of a program is performed in a sequence of phases: The syntactic analysis partitions the program into its elements, like keywords, variables, operators, numbers etc., and builds from these an abstract syntax tree (AST) representing the semantic components of the program, like assignments, loops, case-decisions, procedures or functions. This is followed by a semantic analysis which verifies resp. computes context conditions and type information and augments the AST by corresponding annotations. Finally, the coder generates target (machine) code from the annotated AST (possibly via intermediate program representations). Before and after the actual code generation several optimizations can be applied. The syntactic and symbolic analyses constitute the so-called front-end, code generation and optimizations the back-end of the compiler.

Compiler safety methods typically either try to formally prove the correctness of the translation steps of the compiler or they rely on the testing of the compilation results. Other approaches try e.g. to exclude potential sources of problems like ambiguities in general by restrictions of the source language.

For compilers in the area of automotive applications, frequently used source languages are e.g. C and Simulink/Stateflow, while machine language of embedded processors and again C are typical compiler target languages. We briefly discuss the application of compilers in the automotive context and derive particular requirements in this area.

The deployment of programmable micro controllers in the automotive industry rose rapidly since their introduction in the nineteen-seventies, particularly the last decade, and does not show any sign of a slow-down (see Fig.1). Modern upper-class cars may feature 100 or more individual control units, each equipped with 1-2 micro controllers and up to several MB program memory. These are cross-linked into an overall network via up to 5 bus systems and up to 10 single buses. A single control unit may encompass up to 100k lines of code of software. In addition to the many comfort functions also safety-relevant functions become more and more important, thus also increasing the safety-relevant aspects of software development dramatically. This situation in combination with the typical requirements for automotive embedded software yields the following requirements and general conditions for modern automotive software development:

- Strong real-time requirements, e.g. for the driving-dynamics management and engine control (cycle times < 5ms and jitter < 10 μ s).
- High cost pressure because of large quantities of control units.
- Narrow time frames for the software development because of short model cycles in the automotive segment and the increasing innovation speed.
- High requirements to the integration of the OEMs due to highly distributed software development. Many suppliers participate in the development of hardware and software used in one model. The integration of these in a highly cross-linked architecture is very complex.
- High safety requirements, because failure of embedded software can result in injuries or even death.

This yields requirements for tools used in the software development process and in particular for compilers as follows:

- High demands with respect to optimization of memory and run-time efficiency. As is typical for embedded software, cost reduction is very important because of a high quantity of units.
- Fast availability for existing, but in particular also for new hardware platforms. Because of a high variability of the used processor platforms (with advanced instruction sets and completely new processor architectures, like Infineon TriCore2), the contemporary availability of development tools for these platforms is important for a successful adoption in serial development.
- Usage of ANSI/ISO C as integrating language. The relevant standards in the automotive area and the established development processes established ANSI/ISO C as lingua franca for the integration.
- Usage of established notations. It is difficult to introduce new notations for the modeling of software aside from those used in MATLAB & Simulink, ASCET or MatrixX because the high number of users from many companies complicates the acceptance of new notations and standards.
- While it is possible to ensure the safety of the compilation result by examining the machine code, a safe compiler gives an important benefit: when a new version of a safe compiler is released, testing the output of the new compiler can be reduced.

Methods and techniques for ensuring compiler safety must take these conditions into consideration when used in the automotive context.

Furthermore, as mentioned above, a number of standards, see e.g. [PFP94,DINEN61508] refer to safety of software development and influence, thus, the above criteria.

3. Overview of Relevant Approaches

The investigation of compiler safety can be classified basically into

1. approaches based on formal descriptions of systems and constraints and correctness proofs,
2. test-based approaches, and
3. a priori language restrictions.

We give a brief overview about the various methods and shortly refer to the restricted published results concerning compiler safety for the particular area of automotive applications.

The first class covers the approaches of compiler verification; these aim at a completely verified system. However, the usage of theorem provers requires not only a high user interaction during the actual proof process, but first and foremost a complete formalization of a large software system, that is, the compiler. Since this is not manageable with current tools and methods, current compiler verification approaches usually consider (strongly) restricted (subsets of) programming languages. Other approaches based on formal system descriptions and proof methods are e.g. model checking, abstract interpretation, proof-carrying code, and program result-checkers. In Section 4, we consider compiler verification and other proof-based methods in greater detail and discuss the applicability of these approaches in the automotive context.

Even though there are some approaches based on formal descriptions and proof methods applied in the automotive area, a complete verification of compilers is not yet possible. ClawZ [ACOS00] is a tool which translates Simulink models into a Z notation. Since it is possible to translate also programs of implementing languages like Ada into Z, the idea is to show the equivalence of the Z specifications originating from the model and from the implementation using proof supporting tools. This approach has e.g. been applied to show the correctness of a flight control system. A similar approach with specification language Z and implementation language SPARK is presented in [KA96].

In contrast to the above, test-based techniques do not require changes (or even reimplementations) of the compiler. The complex tasks of a system formalization and a formalization of further conditions and constraints can be dropped and the complicated correctness proof of the transformation is not necessary anymore. However, there is now the problem of establishing an extensive set of test cases with a maximum coverage of program traces. We inspect test-based approaches for the automotive context in Section 5.

In addition to the above two main approaches for compiler safety, there is a third method to be mentioned in this context: Language restrictions aim to constrain the programming language semantics such that unsafe constructs are not possible anymore, which makes it easier to verify programs and their context conditions. These methods are occasionally used in connection with verification based systems.

Representatives of this approach in the automotive context are, the MISRA-C initiative [McC04,MISRA04] which a priori restricts the language C such that ambiguous expressions can be avoided, and SPARK [Ch02,R01,B03], a subset of Ada, but with additional constructs which support a data flow analysis and a partial verification. Gaul et al. [GKC01] investigate the applicability of Java for embedded systems in automotive applications and provide proposals for an adaptation of the language definition for an application of Java in the automotive industries.

In the following sections we investigate the two main directions for compiler safety, namely compiler verification and test-based approaches.

4. Compiler Verification

The goal of compiler verification is a complete proof of correctness for all parts of the compiler. The verification process can be divided into two parts: the first is the verification of the transformation steps involved in the translation process, and the second is the verification of the implementation of these steps. Correctness of the transformations is called *translation correctness*, whereas the other is called *implementation correctness*. The former has the property that it must be proved once for each transformation between a source and a target language, whereas the latter must be proved for each implementation. If, for example, several compilers are written which implement verified translations, each implementation must be proved independently from the others.

There are several ways of proving a compiler correct. It can be implemented with the help of a theorem prover, which only allows the user to write programs which satisfy their specification. Other methods include the construction of correct programs by using only provably correct refinement steps, or by verifying the program code using interpretation techniques. Similar approaches enrich programs with proofs of their correctness or provide result-checkers. Different techniques have also been combined in order to verify programs. We will now look at each of these methods in turn.

The overview report by Glesner [Gle05] gives an introduction to the theory and practice of compiler verification. It treats especially the state of the art with respect to the translation and implementation correctness. The report draws the conclusion that compiler verification of a realistic compiler is not yet feasible. Instead, Glesner proposes an alternative way of proceeding, which will be described below, in the context of proof-carrying code.

4.1. Verification by Proving

Verification of the complex algorithms used in compilers cannot be automated and requires extensive user interaction with a theorem prover. Therefore, verification of large programs (such as compilers) is currently not feasible. Nevertheless, verification techniques and supporting software such as theorem provers have made significant progress over the last years, and parts of compilers have been successfully verified.

An extensive bibliography has been published by Dave [Dav03]. In the following we will summarize some important work on translation correctness and implementation correctness with the assistance of theorem provers.

Translation correctness is about the correctness of the algorithms used in a compiler. The various algorithms employed in a compiler are too complex to be completely verified, but for compilers for more restricted programming languages there exist promising results. Leroy [Ler06] and Leroy et al. [LBD06] have implemented a compiler for the programming language Clight (a subset of the C language) for the PowerPC processor. Leroy [Ler06] describes the (locally optimizing) code generator, and how all translation steps were verified. Leroy et al. [LBD06] deal with the compiler front-end which handles the translation from a rich language with a complex type system to a simpler language. The compiler was implemented in the language of the theorem prover Coq and its correctness was proved by showing the equivalence of the semantics of the various intermediate languages. Leroy reports interesting numbers: the amount of specifications, definitions and proofs is between about 7.5 to 8 times as large as the program code. This shows that the effort required for verification is significant, but Leroy emphasizes that the amount of work does not increase by the same factor, because verification reduces the time necessary for testing and correcting errors.

The VLISP project [GRW95] developed a complete verified compiler and run-time system for the Lisp dialect Scheme. This system was verified by hand without help from proof-assisting software.

Instead of proving implementation correctness, it is possible to construct a verifying compiler – that is, a compiler which verifies that the program's translation is correct. This compiler can be applied to itself in order to get a verified compiler. A design for such a verified compiler was given by Necula [Nec00], who showed how the GNU C Compiler (GCC) could be verified in this way.

Software-assisted program verification is very safe, if enough trust is given to the implementation of the theorem provers. Most authors remark that it is well-known in principle how to verify compilers, but that we lack the engineering capabilities to verify large software systems (e.g., compilers) completely. A major task is, of course, to specify, what a “correct compilation” is, that is, to give a formal description of the task that shall be performed by the compiler.

4.2. Refinement Algebras

In contrast to compiler verification as described above, where the algorithms and implementation of a compiler are proved, another approach is to start with the specification and then to use small transformations on this specification repeatedly, until a program is derived. When each transformation is preserving the meaning of the program, the final result is correct by construction. Müller-Olm [MO97] has shown how to use this approach (named refinement-algebraic approach) for specifying refinement rules for a compiler.

Actually, the principal ideas of this concept date back to the work of program transformation. In the context of the CIP project, Pepper [Pep79] introduced the concept of “transformational semantics”. However, at that time, verification systems of the necessary size were not feasible; therefore the work remained at a theoretical level.

In the work of Müller-Olm [MO97], for example, the processor architecture (the Inmos Transputer) was specified, and the source language TPL as well. Then several intermediate models and the transformations between these models were specified. Based on these, a functional specification for a compiler was derived, which can be implemented in a suitable programming language.

The drawback of refinement algebras is that the effort required for formalization is very large, similar to machine-assisted program verification. Furthermore, the formalizations in the literature have only treated simple languages and simple virtual target machines, not realistic hardware. Conceptually, this approach is promising, because the amount of work required for proving the correctness is divided into small, manageable parts. Unfortunately, not all aspects of a compiler can be divided in such small parts, some problems can only be solved using relatively complicated algorithms and data structures.

4.3. Static Analysis

Many properties of programs can be checked by using an analysis technique called abstract interpretation [CC04]. A tool takes the program, which is to be proved correct, as its input and executes it using a special interpreter. This interpreter uses special operations and value domains for program variables, which are abstractions of real operations and domains. These abstractions limit the possible values each variable can take to finite sets, so that execution of the program terminates. The abstractions are chosen in a way that they reflect some property of the program which is to be verified, so that the result represents whether the verification was successful or not. The advantage of the abstraction is that the verification tool can check the correctness of all possible inputs, the disadvantage is that it introduces some approximation, so that some programs may be rejected, even if they are correct.

Abstract interpretation has been used for verifying compilers by McNerney [McN91], who used a combination of testing and verification. For a set of test programs, the compiler created both optimized and un-optimized binaries; then the equivalence of both versions was shown by abstract

interpretation of the assembler code.

The main advantage of abstract interpretation is that the effort to prove some specific property of a program is not too large. Complete verification of compilers has not yet successfully been done.

The goal of abstract interpretation and other static analysis tools is to prove the absence of certain (run-time) error conditions. This is in contrast to other verification approaches which aim at the complete absence of all errors.

4.4. Proof-carrying Code

Proof-carrying code [Nec97] is a program together with a proof that the program is correct. The compiler constructs this proof as part of its output. When a program is executed, the proof is checked to see whether the program does indeed perform the intended work. The reason for combining the proof with the program is that proof checking is much simpler than proving, and can be done without user interaction.

Necula [Nec97] develops an assembler extension for the TIL compiler for Standard ML, which allows to add hand-written assembler code to an ML program. This is critical, because the assembler code must maintain the data integrity of the compiler-generated code. This is guaranteed by a proof which can be verified when the program is executed.

A variant of proof-carrying code is certificate-based validation [Gle03]. The compiler constructs a certificate which describes which transformation steps have been used in order to generate the target program. Glesner [Gle03] describes how the certification approach was implemented in a compiler back-end. The back-end is based on a rewrite system for the tree-like intermediate program representations. The certificate describes which rewrite-steps have been applied to the representation. The resulting certificates can then be used in conjunction with program result-checking (see below) for ensuring that the resulting code is correct.

In the setting of translation validation [PSS98], the proof-carrying code approach was extended by providing a theoretical foundation based on three aspects: (a) a semantic framework for describing source and target languages, (b) a refinement relation, and (c) an automatic method which produces proofs of the equivalence between the source and the target. The translation from Signal programs to C code is described in this setting. The generated and validated code is then translated by a C compiler without optimizations in order to reduce the risk of bugs in the C compiler optimizer.

The approaches in the literature each perform analysis and validation of different parts of a compiler. The transformations of the internal representation of the compiled programs, which include optimizations, are proved correct. Unfortunately, it has not yet been shown that all steps involved in the compilation process can be formulated as proofs usable in a proof-carrying code system.

4.5. Program Result-Checker

A program result-checker [BK89] is a program component which takes as its input the result of some calculation and checks whether it is correct. This can be done by calculating the result again (using another, proved algorithm and then comparing the results), or, if there exists a suitable test, by making some consistency check on the result. In the original definition, the checking algorithm is required to be in a more efficient complexity class than the original algorithm, to make sure that both components do not use a similar algorithm. Often, the checking algorithm must be a probabilistic algorithm, so that it can stay within these time bounds. The checking algorithm may then produce wrong answers, although the error can be reduced by repeating the test.

There is a close relationship between program result-checkers and compiler verification, in that

several verification approaches work with result-checkers. On the other hand, result-checkers have wider applicability than just program verification. Inspired by error detection and correction in communication protocols, result-checkers have been improved by adding correction facilities. This is possible, for example, in the field of numerical computations, where approximate solutions are expected anyway, and where techniques for increasing the precision exist.

In the context of compilers, approximate results are not allowed, of course, but result-checking is an interesting solution to the verification nevertheless. Consider for example the process of register allocation (assignment of machine registers to program variables) in the code generator of a compiler. A coloring register allocator, which is a complex algorithm based on graph coloring, could be used to calculate a register assignment. When it has produced a result, the result-checker takes the register assignment and checks whether any interferences (assignments of several variables to the same register) have been introduced, and rejects the assignment in that case.

Program result-checkers are a realistic alternative to complete verification. Since checking a result is often much more efficient than calculating it (especially in compilers, where often search algorithms with high computational complexity are involved), adding a result-checker can increase the reliability of a compiler without adding too much to the compile time cost.

Result-checkers complement test-based approaches (which only consider those parts of the compiler for which tests exist) by checking the results for each program run. It is important to note that test-based approaches are normally only used to test while developing the compiler, whereas some verification approaches such as program result-checkers test even after development of the compiler, namely when the compiler is used in the production of software by the client.

As Wasserman and Blum put it: "It is our hope that result-checking may form the basis for a debugging methodology more rigorous than the testing suite and more pragmatic than verification." [WB97, p. 831]

4.6. Model Checking

Model checking is a technique that originated in the verification of hardware components, but is more and more applied to software projects as well. The idea is to first identify a set of erroneous states and then to cover the whole state space, showing that no erroneous states can be reached from the initial configuration. A comprehensive overview is given by Clarke et al. [CGP00]. To deal with combinatorial explosion, several abstraction techniques exist. Nevertheless the state space of a compiler is enormous such that model checking approaches for compilers focus on special parts only, for example the optimization stage [Fre02, SBCJ02].

Model checking is certainly well suited for safety-relevant contexts, but to deal with huge state spaces combinations with traditional verification methods have to be found.

5. Test-based Approaches

Test-based approaches do not prove the correct behavior of an compiler implementation under all circumstances, but show that the compiler gives correct output for a finite (but big) number of test programs. To increase the confidence in a compiler, three problems have to be addressed:

1. How can we obtain a test suite, i.e. a collection of programs to be translated with our compiler under test?
2. For each program in our test suite, we have to check whether it is compiled correctly, and therefore need a way to define the expected output of our compiler, i.e. a test oracle.
3. How can we measure whether enough relevant test cases have been executed? To answer

this question, we need to define so-called coverage criteria.

A survey of several approaches can be found in [KP05]. The literature also gives ways to compute error probabilities considering the number of tests run (see e.g. Ehrenberger [Ehr02]), however, it is not clear how to apply this to the special case of compilers.

5.1. Compliance Suites

Compliance suites are designed to test whether a compiler matches the specification of a language. They typically provide a set of manually written programs, each designed to cover a specific part of the specification text, often just a single sentence or a short fragment. Additionally, they define the expected behavior of the test programs according to the interpretation of the standard by the test suite creator.

There are compliance suites for numerous programming languages. The design of the Ada compiler validation suite is described by Goodenough in [Goo80], pointing out several features contributing to the success of the Ada suite:

1. the creation of the test suite in parallel of the development of the language standard,
2. the availability of a compiler implementor's guide mentioning potential pitfalls,
3. different classes of test cases, and
4. the standardized process of testing a compiler.

To cover not only every feature of a programming language by itself but several combinations of features, Mandl [Man85] proposes the orthogonal latin squares method. It avoids combinatorial explosion by avoiding redundant tests while maintaining a good coverage of feature combinations.

The process of Ada compiler validation is detailed in the international standard ISO/IEC 18009:1999 [Int99]. It distinguishes testing laboratories and a central monitoring system that maintains the test suite and supervises the testing laboratories.

While the compliance suite for Ada is often regarded as the most successful suite, validation suites for other languages exist, such as the suites for the C language of Plum Hall [Plu] and Perennial [Per]. A comparison of compliance suites for several languages is given by Tonndorf [Ton99], highlighting the standardization of the Ada suite and the institution of supervised test laboratories as important advantages of the Ada suite.

The suite for the Ada language, the process of its creation in parallel to the development of the language standard, and the standardization of the process of testing a compiler are regarded as well designed and appropriate for compilers in safety-relevant areas. Nevertheless, test suites cannot prohibit all compiler problems, it is of course possible to compile a huge test suite without errors but failing on real world programs. One also has to be aware of another problem: when “official” test suites exist, compiler writers tend to “customize” their compilers to those suites. Then the validation by passing the test suite becomes more and more irrelevant. This problem can be addressed by properly maintaining and extending the suite.

5.2. Automatic Generation of Test Cases

The work of Kalinov et al. [KKPS02, KKP+03a, KKP+03b] studies the automatic generation of test cases in the context of the language mpC, a dialect of C equipped with special constructs for parallel processing. The constructs of mpC are defined using Abstract State Machines (ASMs) by a set of rules. These rules can be used on the one hand to generate valid test cases, and on the other hand to derive the expected behavior of translated programs. A similar approach is described by Esin et al. [ENZ04] but with emphasis on looping constructs and conditional expressions.

Another line of research focuses on the optimization stage of a compiler, which is an essential component of compilers for embedded systems. Burgess et al. [BS96] generate Fortran programs that contain especially many fragments that might be optimized by the compiler by e.g. common subexpression elimination, algebraic simplification, or dead code elimination. As a test oracle they simulate the behavior of the test program during generation and place comparisons with the results of the simulation at adequate positions into the test programs. Kossatchev et al. [KPZZ03] propose a more systematic technique to analyze optimization algorithms, extracting a test program generator from this analysis.

It is difficult to obtain reliable test oracles for automatically generated test programs: even with a formal definition at hand (which is rarely the case for real world programming languages), the interpreter of the formalized definition may contain similar errors as the compiler, resulting in errors that cannot be found by testing. The advantage of the automatic generation of test cases is the possibility to easily construct a huge number of test cases. Additionally, random test cases can cover situations not foreseen by test engineers.

5.3. Back To Back Testing

Back to back testing is a technique dealing with the test oracle problem, i.e. how to obtain the expected results of a compiler. The idea is to translate a test program twice, the first time with the compiler under test, the second time using a trusted reference implementation. The reference implementation might be provided with the language standard, might be a very simple and thus inefficient but easy to verify implementation, or might be the same compiler but with e.g. all optimizations turned off for testing the optimization stage of the compiler under test. A study focusing on a compiler for digital signal processors (DSPs) has been published by Popovic et al. [PKT00]. Jaramillo et al. [JGS99, JGS02] propose a way to not only detect but to locate errors in the compiled code. Yoshikawa et al. [YSO03] describe the test of a Java just in time (JIT) compiler via automatically generated code, taking Sun's Java compiler as a reference implementation.

All back to back approaches rely crucially on the correctness of the chosen reference implementation. The reliability can be enhanced by choosing an independent implementation as a reference instead of merely turning the optimization off. But back to back testing is the only way of testing randomly generated test cases.

5.4. Design by Contract

Design by contract aims to enhance the reliability of software by defining pre- and postconditions of subroutines and invariants holding during the whole runtime. These so-called contracts can be checked at suitable places in the tested program. Gibbs et al. [GMP02, GM03] apply this approach to a compiler front-end for the C++ language.

Design by contract is not a complete testing process on its own, the problem how to generate sufficient and relevant test programs remains. A strength of this approach is that errors in the compiler may be found more easily since sufficient knowledge is given, which contract is violated at which parts of the compiler, as opposed to signaling only that a certain program is translated incorrectly.

5.5. Testing Model Transformations and Code Generators

Model transformation engines and code generators translate visual system models into either new visual models with a lower degree of abstraction or into conventional programming language texts. While these tools have much in common with traditional compilers, they have their own challenges. An overview of these challenges is given by Benoit et al. in [BDTM+06].

Steely et al. [SL04] and Yuehua et al. [LZG05] describe approaches to test model transformations. However, both approaches need much manual effort.

The work of Stürmer et al. [SC03, BKS04] concentrates on code generators. They generate test cases automatically via graph grammars that formally specify the code generator under consideration.

Baudry et al. [BM06] deal with the problem of how to cover all relevant parts of a model transformation engine. They analyze a given test suite by creating several faulty variants of the transformation engine, so called mutants. The more mutants are detected by running the test suite, the better the transformation engine is covered.

Visual tools gain more and more popularity. To improve their reliability, promising techniques are proposed, but the available results are not as well-grounded as for traditional programming languages.

6. Conclusion

This paper analyses the status quo of the science and technology for compiler safety methods. We classified the wide spectrum of general methods and systematized the approaches in particular under the aspect of the automotive segment. We discussed in detail compiler verification and testing methods as the main directions in compiler safety methods.

Currently, the verification methods and techniques usually consider only partial aspects of compilation or particular language features. While, in general, all these methods used together would be sufficient to get a completely correct certifiable compiler, there remain two open problems: on the one hand, it is today neither clear nor investigated, whether it is possible to combine or integrate the discussed approaches. On the other hand, the scaling problem in particular with respect to verification approaches is largely open: as discussed in the above sections, the verification approaches have been investigated typically for restricted case studies, that is for small prototypic compilers or generators, while the application in practical automotive systems needs a transfer of the results onto real languages and programs.

Test methods are more encouraging. The Ada test suite has shown that industrial-strength compilers can be thoroughly tested using manually constructed test cases. These suites could be complemented using automatic test generation methods which cover unforeseen corner cases.

It should be noted that verification as well as testing exhibit a similar problem: One needs to specify, what the compiler shall do, respectively, what the test cases should yield.

In conclusion, we think that to improve the reliability of compilers for safety-related applications in the automotive industry a combination of test-based approaches is the most promising way.

7. References

- [ACOS00] R. D. Arthan, P. Caseley, Colin O'Halloran, Alf Smith: ClawZ: Control Laws in Z. In: Third IEEE International Conference on Formal Engineering Methods ICFEM. Pages 169-176, 2000.
- [B03] John Barnes: High Integrity Software: The SPARK Approach to Safety and Security. Praxis Critical Systems Limited (Editor). Addison Wesley. 2003.
- [BDTM+06] Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model transformation testing challenges. In ECMDA workshop on Integration of Model Driven Development and Model Driven Testing, Bilbao, Spain, July 2006.

- [BK89] M. Blum and S. Kanna. Designing programs that check their work. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of Computing*, pages 86-97, New York, NY, USA, 1989. ACM Press.
- [BKS04] Paolo Baldan, Barbara König, and Ingo Stürmer. Generating test cases for code generators by unfolding graph transformation systems. In *Proc. of ICGT '04 (International Conference on Graph Transformation)*, pages 194—209. Springer-Verlag, 2004. LNCS 3256.
- [BM06] Benoit Baudry and Jean-Marie Mottu. Mutation analysis testing for model transformations. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA 06), Bilbao, Spain, 2006*.
- [BS96] C. J. Burgess and M. Saidi. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology*, 38:111-119, 1996.
- [CC04] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359—366. Kluwer Academic Publishers, 2004.
- [CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.
- [Ch02] Rod Chapman: MISRA-C at SIL4? Perspectives and Alternatives. Talk, SAE Embedded Software Presentation Series. <http://www.praxis-his.com/sparkada/misra.asp>, 2002.
- [Dav03] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2-2, 2003.
- [DIN-EN-61508] Deutsches Institut für Normung: DIN EN 61508.
- [Ehr02] W. Ehrenberger. *Software Verifikation*. Carl Hanser Verlag München Wien, 2002.
- [ENY04] Anton Esin, Andrey Novikov, and Rostislav Yavorskiy. *Experiments on Semantics Based Testing of a Compiler*. online manuscript, 2004.
- [Fre02] Carl Christian Frederiksen. Correctness of classical compiler optimizations using CTL. In Jens Knoop and Wolf Zimmermann, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.
- [GKC01] Thilo Gaul, Antonio Kung, Jerome Charousset: AJACS: Applying Java to Automotive Control Systems. In: Caspar Grote, Renate Ester: Proceedings of Embedded Intelligence Conference. Pages 425-434. 2001.
- [Gle03] Sabine Glesner. Program Checking with Certificates: Separating Correctness-Critical Code. In *FME Symposium*, volume 2805 of *Lecture Notes in Computer Science*, pages 758-777. Springer Verlag, 2003.
- [Gle05] Sabine Glesner. Optimierende Compiler: Vertrauen ist gut, Verifikation ist besser! Technical Report 2005-28, Universität Karlsruhe, September 2005.
- [GM03] Tanton H. Gibbs and Brian A. Malloy. Weaving aspects into C++ applications for validation of temporal invariants. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 249, Washington, DC, USA, 2003. IEEE Computer Society.
- [GMP02] Tanton H. Gibbs, Brian A. Malloy, and James F. Power. Automated validation of class invariants in C++ applications. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK, 2002*.
- [Goo80] John B. Goodenough. The Ada compiler validation capability. In *Proceedings of the ACM-SIGPLAN symposium on The ADA programming language*, pages 1-8, 1980.

- [GRW95] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. Vliisp: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1-2):5-32, 1995.
- [Int99] International Organization for Standardization. *ISO/IEC 18009:1999: Information technology - Programming languages - Ada: Conformity assessment of a language processor*. International Organization for Standardization, 1999.
- [JGS99] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Comparison checking: an approach to avoid debugging of optimized code. *SIGSOFT Softw. Eng. Notes*, 24(6):268-284, 1999.
- [JGS02] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Debugging and testing optimizers through comparison checking. *Electr. Notes Theor. Comput. Sci.*, 65(2), 2002.
- [KA96] D. J. King, R. D. Arthan: Development of Practical Verification Tools. The ICL Systems Journal. Vol.11, No.1, 1996.
- [KKP+03a] A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Coverage-driven automated compiler test suite generation. *Electronic Notes in Theoretical Computer Science*, 82(3), 2003.
- [KKP+03b] A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM specifications for compiler testing. In *Abstract State Machines*, 2003.
- [KKPS02] A. Kalinov, A. Kossatchev, M. Posypkin, and V. Shishkov. Using ASM specification for automatic test suite generation for mpC parallel programming language compiler. In P. Mosses, editor, *Proceedings of the Fourth International Workshop on Action Semantics, AS 2002*, number NS-02-8 in BRICS Notes Series, pages 99-109. University of Aarhus, Department of Computer Science, 2002.
- [KP05] Alexander Kossatchev and Mikhail Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10-19, 2005.
- [KPZZ03] A. S. Kossatchev, A. K. Petrenko, S. V. Zelenov, and S. A. Zelenova. Application of Model-Based Approach for Automated Testing of Optimizing Compiler. In *Proceedings of the International Workshop on Program Understanding*, 2003.
- [LBD06] Xavier Leroy, Sandrine Blazy, and Zaynah Dargaye. Formal verification of a C compiler front-end. In *Proceedings of Formal Methods 2006 (FM'06)*, 2006.
- [Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42-54, New York, NY, USA, 2006. ACM Press.
- [LZG05] Yuehua Lin, Jing Zhang, and Jeff Gray. A Framework for Testing Model Transformations. In *Model-driven Software Development*. Springer, 2005.
- [Man85] Robert Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. *Commun. ACM*, 28(10):1054-1058, 1985.
- [McC04] Gavin McCall: Introduction to MISRA-C. Talk, Visteon Corporation. <http://www.misra-c2.com/>, 2004.
- [McN91] Timothy S. McNerney. Verifying the correctness of compiler transformations on basic blocks using abstract interpretation. In *PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 106-115, New York, NY, USA, 1991. ACM Press.
- [MISRA04] MISRA: MISRA-C:2004 - Guidelines for the use of the C language in critical systems,

ISBN 095241623.

- [MO97] Markus Müller-Olm. *Modular Compiler Verification - A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer, 1997.
- [Nec97] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106-119, New York, NY, USA, 1997. ACM Press.
- [Nec00] George C. Necula. Translation validation for an optimizing compiler. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83-94, New York, NY, USA, 2000. ACM Press.
- [Pep79] P. Pepper. A study on transformational semantics. PhD Thesis. TU München, 1979.
- [Per] Perennial, Inc. CVSA - Perennial C Compiler Validation Suite. Website.
- [PFP94] Shari Lawrence Pfleeger, Norman Fenton, Stella Page: Evaluating software engineering standards. *Computer*, Vol.27, No.9, pages 71-79. IEEE Computer Society Press. 1994.
- [PKT00] M. Popovic, V. Kovacevic, and M. Temerinac. Software Testing Concept Used for MAS/C-Compiler. In *26th EUROMICRO 2000 Conference, Informatics: Inventing the Future, 5-7 September 2000, Maastricht, The Netherlands*, pages 2224-, 2000.
- [Plu] Plum Hall, Inc. The Plum Hall Validation Suite for C. Website.
- [PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In B. Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pages 151-166. Springer Verlag, 1998.
- [R01] C. Roderick: SPARK – A state-of-the-practice approach to the common criteria in implementation requirements. 2nd International Common Criteria Conference. 2001.
- [SBCJ02] K. C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Geometric Model Checking: An Automatic Verification Technique for Loop and Data Reuse Transformations. In *Proceedings of the COCV-Workshop (Compiler Optimization meets Compiler Verification)*, volume 65. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 5th European Conferences on Theory and Practice of Software (ETAPS 2002), 2002.
- [SC03] Ingo Stürmer and Mirko Conrad. Test suite design for code generation tools. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pages 286-290, 2003.
- [SC04] Ingo Stürmer, Mirko Conrad: Code Generator Certification: A Testsuite-oriented Approach. In: *Proceedings of Automotive - Safety & Security*. 2004.
- [SL04] Jim Steel and Michael Lawley. Model-based test driven development of the tefkat model-transformation engine. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France*, pages 151-160, 2004.
- [Ton99] Michael Tonndorf. Ada conformity assessments: a model for other programming languages? In *SIGAda '99: Proceedings of the 1999 annual ACM SIGAda international conference on Ada*, pages 89-99, New York, NY, USA, 1999. ACM Press.
- [WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826-849, 1997.
- [YSO03] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. Random Program Generator

for Java JIT Compiler Test System. In *3rd International Conference on Quality Software (QSIC 2003)*, 6-7 November 2003, Dallas, TX, USA, pages 20-, 2003.