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

Pierre R. Mai
PMSF IT Consulting

Stefan-Alexander Schneider
BMW Group

Automotive 2008
Compiler safety is important to ensure safe software in the automotive context.

What is a compiler? A *compiler* or *code-generator* translates human-readable programs into machine-executable programs.

Why is compiler safety important? When the compiler produces incorrect machine-code, even correct human-readable programs may produce wrong results or crash.
Compiler structure
Compiler Overview

Compiler structure

Compiler

Frontend
Compiler Overview

Compiler structure

- Compiler
  - Frontend
  - Backend

Safety of Compilers and Translation Techniques
Compiler Overview

Compiler structure

- Frontend:
  - Scanner
  - Parser
  - Checker

- Backend

Safety of Compilers and Translation Techniques 3/17
Compiler Overview

Compiler structure

Compiler

Frontend
Scanner Parser Checker

Backend
Compiler Overview

Compiler structure

Compiler

Frontend

Scanner
Parser
Checker

Backend

Opt.
Codegen.
Compiler Overview

Compiler structure

`Compiler

Frontend

Scanner -> Parser -> Checker

Backend

Opt. -> Codegen.`
Compiler Overview

Compiler structure

Compiler

Frontend

Scanner → Parser → Checker

Backend

Compiler Overview

Compiler structure

Compiler Overview

Compiler structure

Compiler structure

- **Frontend**
  - Scanner
  - Parser
  - Checker

- **Backend**
  - Opt.
  - Codegen.

Compiler Overview
Modern upper-class cars feature $\geq 100$ control units with 1–2 processors each.

Program quality is extremely important:
- Many programs are safety-critical,
- Software updates are difficult and expensive,
- Strong real-time requirements.

Fast availability for existing and new hardware platforms.
Modern upper-class cars feature $\geq 100$ control units with 1–2 processors each.

Program quality is extremely important:
- Many programs are safety-critical,
- Software updates are difficult and expensive,
- Strong real-time requirements.

Fast availability for existing and new hardware platforms.

Source languages are e.g. C and Simulink/Stateflow,

Target languages are machine code of embedded processors or C,

Program development is influenced by standards [PFP94, Deu].
We identified three classes of compiler safety approaches:

**Language Restrictions**  
Restrict implementation languages

**Verification**  
Prove program correctness

**Testing**  
Test resulting programs to detect errors
Avoid using dangerous programming language features.

Examples of problem cases:
- Error-prone constructs
- Undefined behaviour
- Ambiguous expressions

Solutions:
- Design style guides which define forbidden constructs
- Use tools which check for conformance
- Implement compilers for restricted languages

Examples:
Spark-ADA [Bar03, Rod01], subsets of C e.g. [Cha02, McC04, LPP05]
Verification

Establish correctness by formal methods.

- Make use of tools to mechanically check correctness.

- Requirements:
  - a precise formal specification of what *correct* means [Dav03],
  - a complete formalization of the compiler,
  - (often) high user-interaction during the actual proof process.

Thus, language restrictions.
Verification by Proving

Formally prove correctness of each translation step

Safety of Compilers and Translation Techniques 8/17
Verification by Proving

Formally prove correctness of each translation step
Optionally machine-check proofs using theorem provers
Verification by Proving

Formally prove correctness of each translation step
Optionally machine-check proofs using theorem provers
Combine proofs to prove correctness of complete compiler
Formally prove correctness of each translation step
- Optionally machine-check proofs using theorem provers
- Combine proofs to prove correctness of complete compiler
- **Drawback**: Requires formal specification of each *intermediate translation result*
Proof-carrying Code

- Generate proof of correctness during compilation and combine with machine program [Nec97]
- On execution, check proof to ensure correctness
- **Drawback:** Automatic proof generation is not possible for general programs
Program Result-Checking

Compiler

Frontend
- Scanner
- Parser
- Checker

Backend
- Opt.
- Codegen

Result Checker
- yes/no

Additional component: Result-Checker [BK89, WB97]
Other Verification Approaches

- **Refinement algebras:** Start with the specification and use small transformations on it repeatedly, until a program is derived. When each transformation is preserving the meaning of the program, the final result is correct by construction [MO97].

- **Static analysis:** Abstract interpretation of certain program properties [CC04, McN91].

- **Model checking:** Automatic generation and testing of the complete (but finitely representable) state space [CGP00, AVA08].
Testing

Establish correctness by testing and validation, i.e. show that the compiler gives correct output for a finite (but big) number of test programs.

1. Compilation of multiple source files,
2. Execution of resulting programs, and
3. Comparison of the actual program behavior and the expected behavior.
Establish correctness by testing and validation, i.e. show that the compiler gives correct output for a finite (but big) number of test programs.

1. Compilation of multiple source files,
2. Execution of resulting programs, and
3. Comparison of the actual program behavior and the expected behavior.

Main problems:

- How to obtain a test suite?
- How to define the *expected* behaviour of the compiler?
- What are appropriate coverage criteria?
Compliance suites

Test suites: Collection of (manually written) test programs, each covering specific compiler feature.

- Automated compilation.
- Automated execution of compiled programs.
- Automated comparison of actual and expected behaviour.

- Examples: Ada [Int99, Goo80], C [PH], comparison of test suites [Ton99]
Compiling twice, once with compiler under test, once with trusted reference implementation.
Back to Back Testing

- Compiling twice, once with compiler under test, once with trusted reference implementation.
- Run both resulting programs.
Back to Back Testing

- Compiling twice, once with compiler under test, once with trusted reference implementation.
- Run both resulting programs.
- Compare both program’s behavior.
Back to Back Testing

- Compiling twice, once with compiler under test, once with trusted reference implementation.
- Run both resulting programs.
- Compare both program’s behavior.

- Advantage: Testing of randomly generated test cases is possible.
- **Problem:** Ensure correctness of the reference implementation.
Define pre- and postconditions for each procedure/method.

Each procedure/method tests its pre-/postconditions.
Define pre- and postconditions for each procedure/method. Each procedure/method tests its pre-/postconditions. Advantage: Compiler errors can be found more easily. Problem: Requires heavy modification of compiler.
Other Testing Approaches

- **Automatic generation of test cases,**
  e.g. [KKP+03a, KKP+03b, ENY04, BS96].

- **Testing model transformers and code generators:** New challenges and promising results [BDTM+06].

- **Model Checking:** Automatic generation and testing of the complete (but finitely representable) state space [CGP00, AVA08].
Conclusion

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

- Language restrictions are already successfully used.
- Verification methods are not yet usable for real-life scenarios (formalization of the compiler, high user-interaction etc.).
- Testing is not as complete as verification, but feasible today.

Combination of test-based approaches is currently the most promising way for ensuring compiler safety in the automotive context.
[AVA08] AVACS – Automatic Verification and Analysis of Complex Systems.  

[Bar03] John Barnes.  
High Integrity Software: The SPARK Approach to Safety and Security.  

[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.

Designing programs that check their work.  

The automatic generation of test cases for optimizing Fortran compilers.  
[CC04] P. Cousot and R. Cousot. 
Basic Concepts of Abstract Interpretation, pages 359–366. 

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 
Model Checking. 

[Cha02] Rod Chapman. 
MISRA-C at SIL4? Perspectives and Alternatives. 

[Dav03] Maulik A. Dave. 
Compiler verification: a bibliography. 

[Deu] Deutsches Institut für Normung. 
DIN EN 61508.

Experiments on Semantics Based Testing of a Compiler. 
online manuscript, 2004.


[McC04] Gavin McCall.
Introduction to MISRA-C.

Verifying the correctness of compiler transformations on basic blocks using abstract interpretation.

Modular Compiler Verification - A Refinement-Algebraic Approach Advocating Stepwise Abstraction, volume 1283 of Lecture Notes in Computer Science.

Proof-carrying code.


