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

Compilers and Compiler Requirements

> 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.

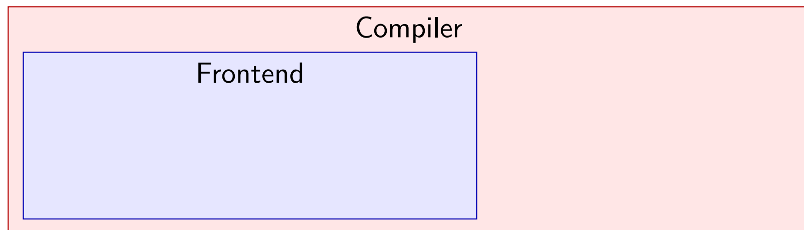
Why is compiler safety important? When the compiler produces incorrect machine-code, even correct human-readable programs may produce wrong results or crash.

Compiler Overview

Compiler structure

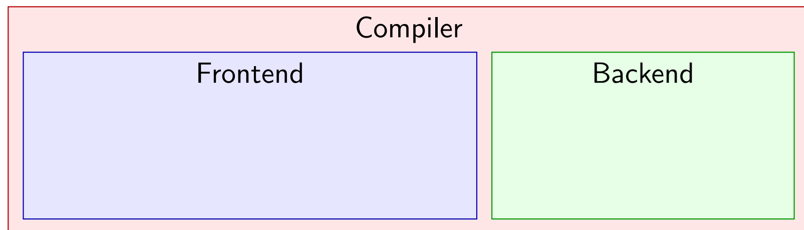
Compiler Overview

Compiler structure



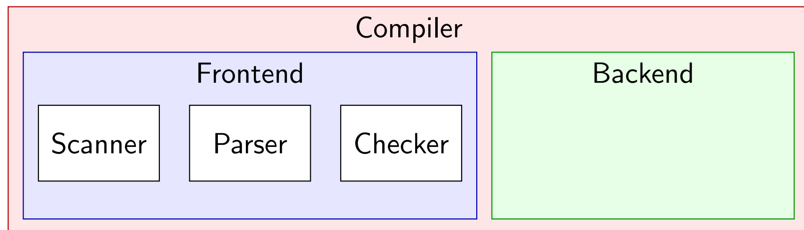
Compiler Overview

Compiler structure



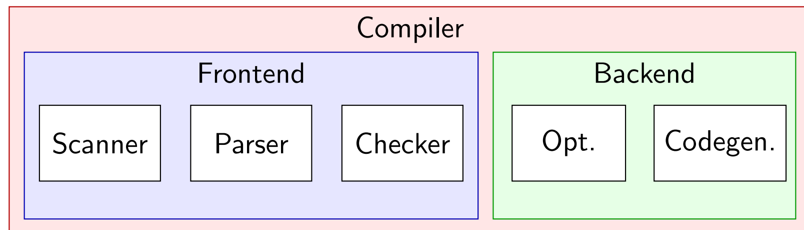
Compiler Overview

Compiler structure



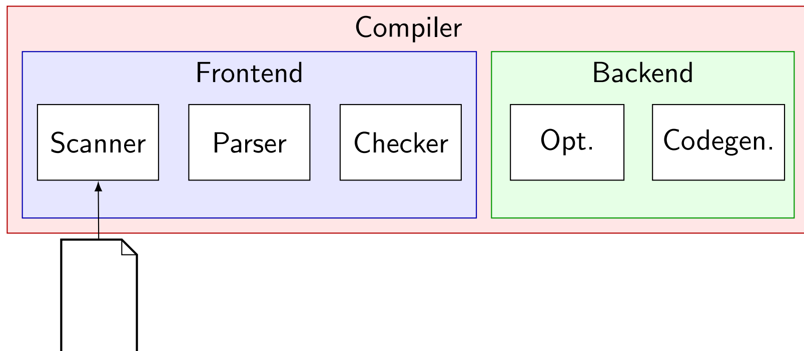
Compiler Overview

Compiler structure



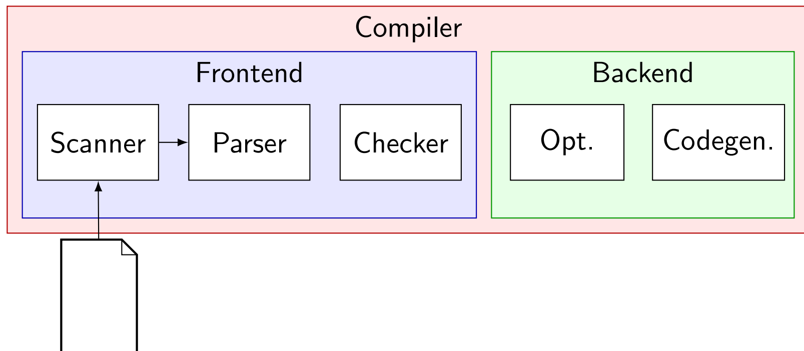
Compiler Overview

Compiler structure



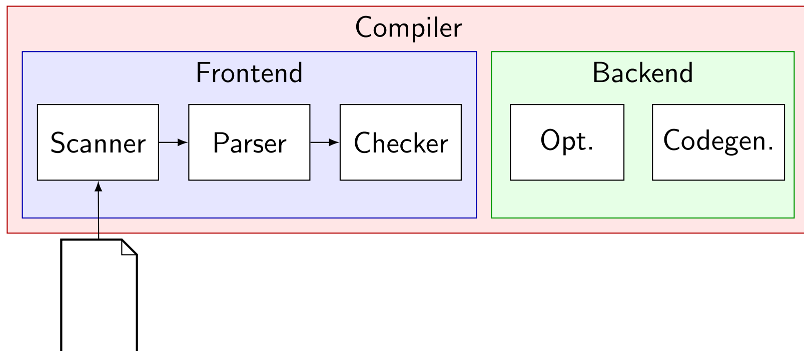
Compiler Overview

Compiler structure



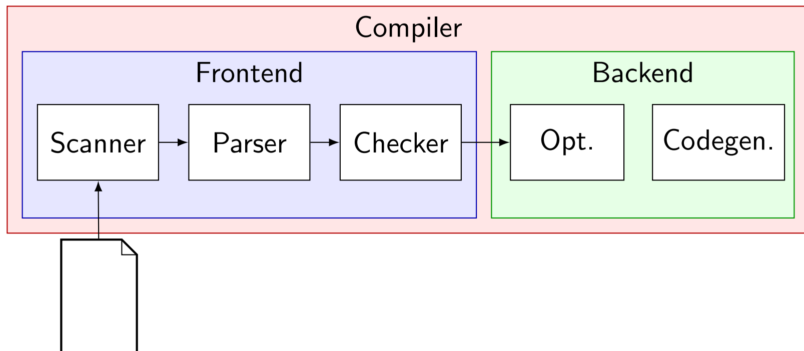
Compiler Overview

Compiler structure



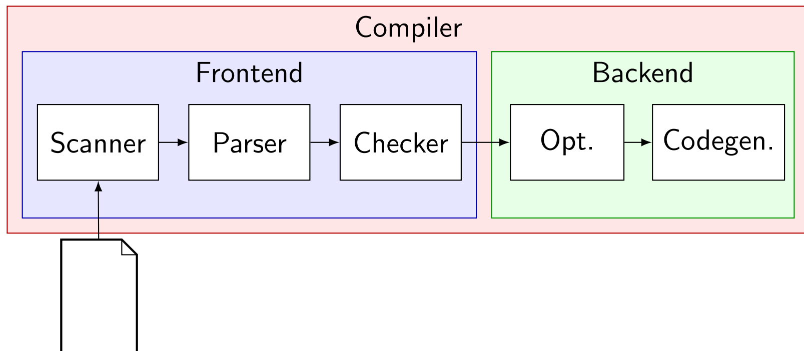
Compiler Overview

Compiler structure



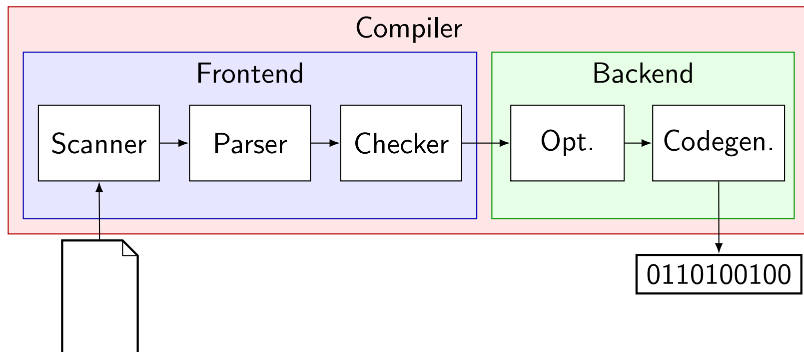
Compiler Overview

Compiler structure



Compiler Overview

Compiler structure



Compilers in the Automotive Context

- 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.

Compilers in the Automotive Context

- 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].

Approaches to Compiler Safety

Goal: ensure that compilers produce correct code.

We identified three classes of compiler safety approaches:

Language Restrictions Restrict implementation languages

Verification Prove program correctness

Testing Test resulting programs to detect errors

Language Restrictions

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]

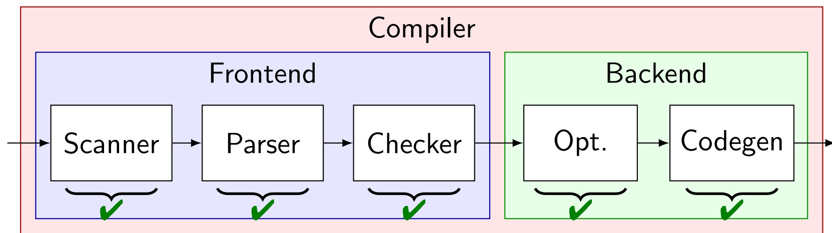
Verification

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.

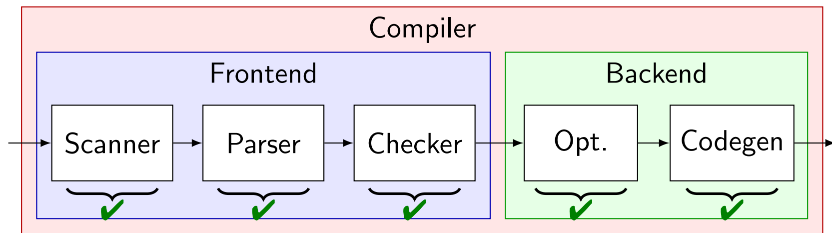
$\rightsquigarrow$ Thus, language restrictions.

Verification by Proving



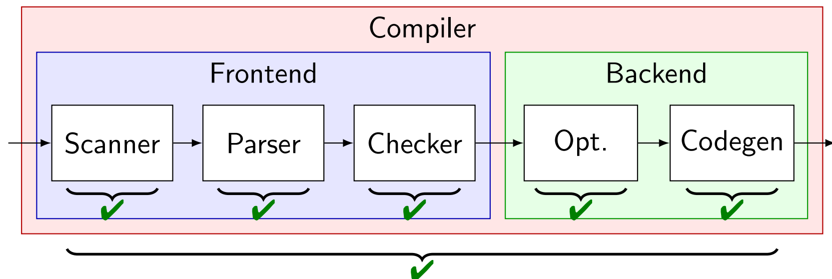
- Formally prove correctness of each translation step

Verification by Proving



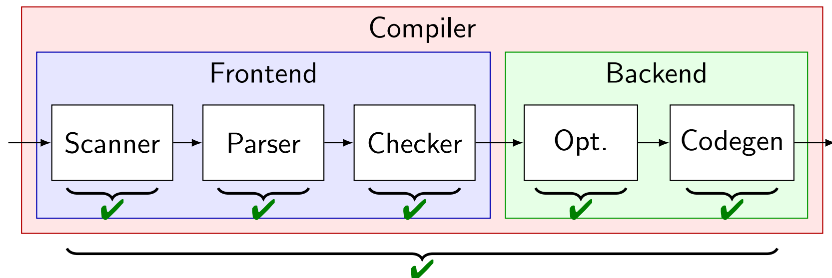
- Formally prove correctness of each translation step
- Optionally machine-check proofs using theorem provers

Verification by Proving



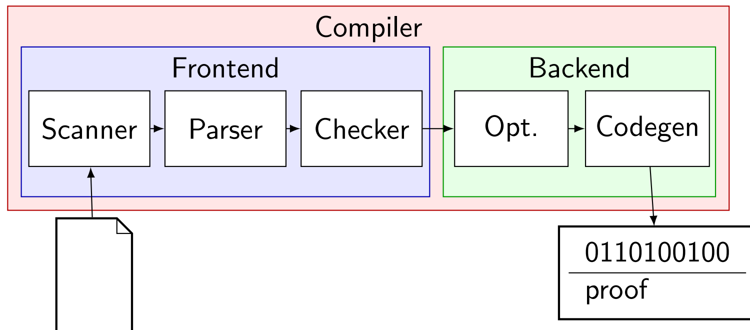
- Formally prove correctness of each translation step
- Optionally machine-check proofs using theorem provers
- Combine proofs to prove correctness of complete compiler

Verification by Proving



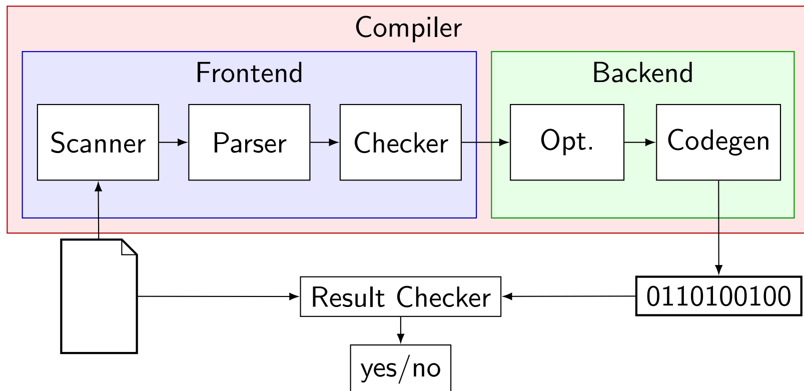
- Formally prove correctness of each translation step
- Optionally machine-check proofs using theorem provers
- Combine proofs to prove correctness of complete compiler
- Drawback: Requires formal specification of *each intermediate translation result*

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



- 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.

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.

Main problems:

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

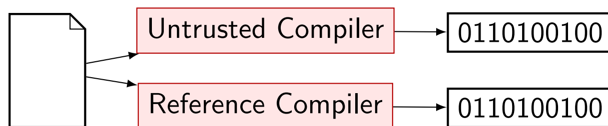
Compliance suites

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.

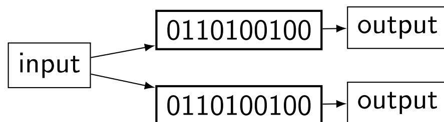
- Examples: Ada [Int99, Goo80], C [PH], comparison of test suites [Ton99]

Back to Back Testing



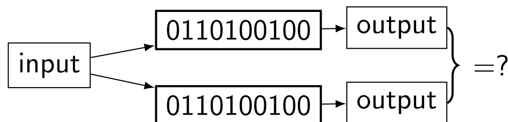
- Compiling twice, once with compiler under test, once with trusted reference implementation.

Back to Back Testing



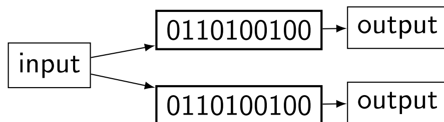
- Compiling twice, once with compiler under test, once with trusted reference implementation.
- Run both resulting programs.

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.

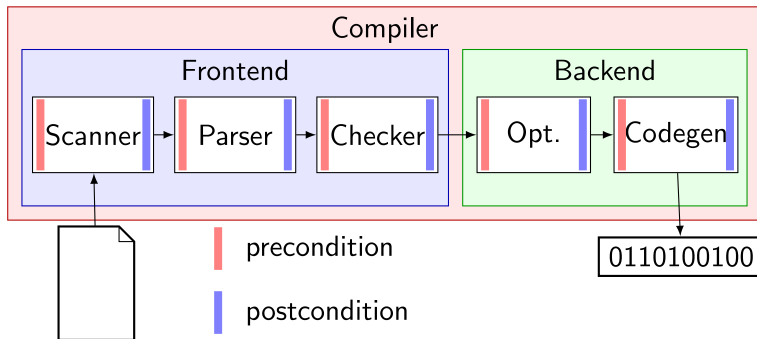
Back to Back Testing



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

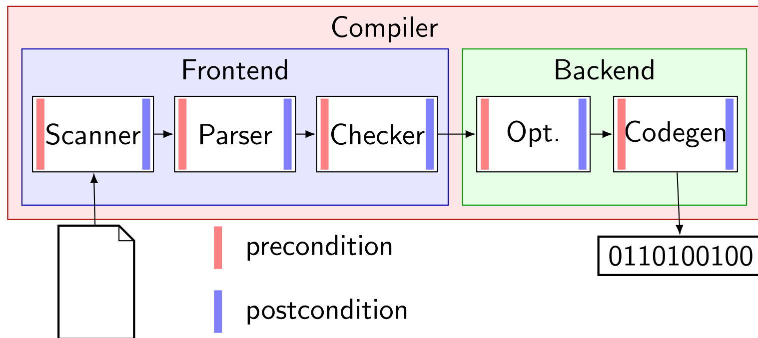
- Advantage: Testing of randomly generated test cases is possible.
- Problem: Ensure correctness of the reference implementation.

Design by Contract



- Define pre- and postconditions for each procedure/method.
- Each procedure/method tests its pre-/postconditions.

Design by Contract



- 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 transformators 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.
`http://www.avacs.org/`, 2008.
last visited 2008-13-11.

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

[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.

[Cha02] Rod Chapman.
MISRA-C at SIL4? Perspectives and Alternatives.
Vortrag: SAE Embedded Software Presentation Series, 2002.

[Dav03] Maulik A. Dave.
Compiler verification: a bibliography.
SIGSOFT Softw. Eng. Notes, 28(6):2–2, 2003.

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

[ENY04] Anton Esin, Andrey Novikov, and Rostislav Yavorskiy.
Experiments on Semantics Based Testing of a Compiler.
online manuscript, 2004.

[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.

[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.

[KKP+03a] Alexey Kalinov, Alexander Kossatchev, Alexandre Petrenko, Mikhail Posypkin, and Vladimir Shishkov.
Coverage-driven Automated Compiler Test Suite Generation.
Electr. Notes Theor. Comput. Sci., 82(3), 2003.

[KKP+03b] Alexey Kalinov, Alexander Kossatchev, Alexandre Petrenko, Mikhail Posypkin, and Vladimir Shishkov.
Using ASM Specifications for Compiler Testing.
In *Abstract State Machines*, page 415, 2003.

[LPP05] D. Leinenbach, W. Paul, and E. Petrova.
Towards the formal verification of a C0 compiler: Code generation and implementation correctness.
In *Proceedings of the 3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*, 2005.

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

[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.

[PFP94] Shari Lawrence Pfleeger, Norman Fenton, and Stella Page.
Evaluating software engineering standards.
Computer, 27(9):71–79, 1994.

[PH] Inc. Plum Hall.
The plum hall validation suite for c.
last visited 2008-11-13.

[Rod01] C. Roderick.
Spark – a state-of-the-practice approach to the common criteria in Implementation requirements, 2001.

[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.