Verification of Optimizing Compilers

Sabine Glesner

Software Engineering for Embedded Systems

Technical University of Berlin

PES

Software bugs are expensive:

- **Mars Climate Orbiter Sonde (1999):**
  - Convertion of non-metric to metric units: numbers forgotten
  - results into loss of sonde
- **Ariane-5 crash (1996):**
  - caused in the end by the convertion of a 64-bit floating point number into a 16-bit signed integer number
- **Pentium bug (1994):**
  - certain divisions lead to wrong result
  - costs Intel nearly 500 million dollar
  - since then: formal verification of floating point algorithms at Intel
- **Formal methods, in particular formal verification, to avoid financial loss**

special cases cause bugs

Overview

- Reasons and prerequisites for formal verification
- Verification of program / system transformations
- Three formal verifications as example:
  - Verification of optimizing compilers
  - Verification of model transformations
  - VATES: Verification of satellite software
- Overview about further research projects
- Conclusions and perspectives

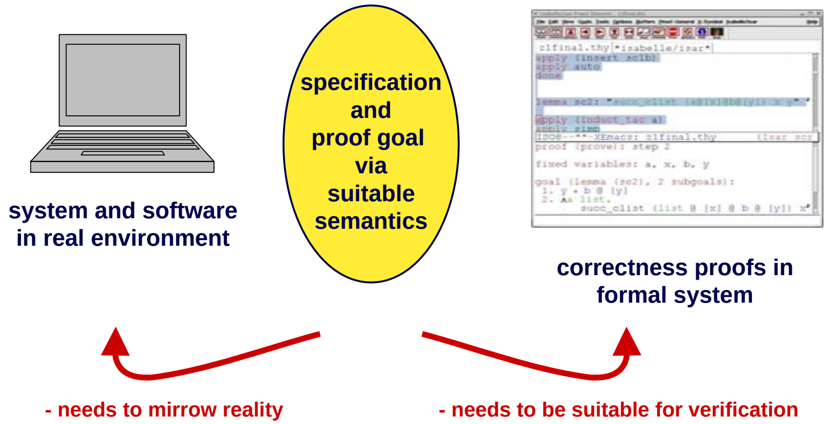
Overview

- Reasons and prerequisites for formal verification
- Verification of program / system transformations
- Three formal verifications as example:
  - Verification of optimizing compilers
  - Verification of model transformations
  - VATES: Verification of satellite software
- Overview about further research projects
- Conclusions and perspectives

Why formal verification?

- **Software systems:**
  - behave exclusively according to formal rules
- **Test and validation:**
  - observing a system for "typical" inputs
  - does not rule out mistakes
- **Verification:**
  - proves allquantified statements
    - "for all conceivable states, it holds that …"
  - **formal verification** with a theorem prover
    - rules out bugs completely
    - is **very** expensive
    - **can nevertheless be worth the extra effort**

Requirements for formal verification



Overview

- Reasons and prerequisites for formal verification
- Verification of program / system transformations
- Three formal verifications as example:
  - Verification of optimizing compilers
  - Verification of model transformations
  - VATES: Verification of satellite software
- Overview about further research projects
- Conclusions and perspectives

Software / system transformations …

- do take place very often:
  - model transformations (e.g. UML to Java)
  - software reengineering
  - in compilers
  - hardware synthesis
  - …
- need to be correct

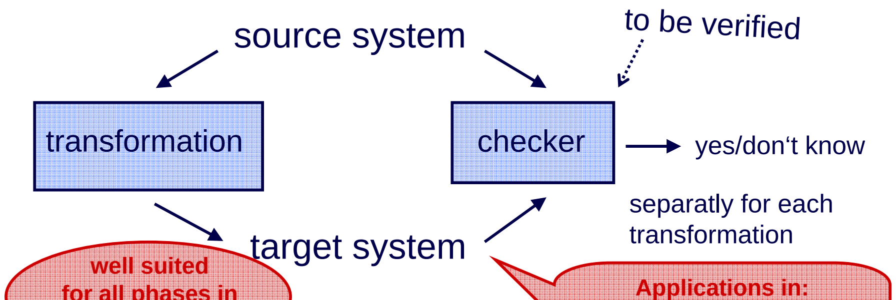
Correctness of transformations

- **Translation correctness**
  - Is the translation algorithm correct?
    - Does it preserves the semantics during transformation?
    - semantics = e.g. observable behavior
  - proof technique: mostly refinement proofs
- **Implementation correctness**
  - Is the translation algorithm correctly implemented?

[Glesner,Goos,Zimmermann, it 46(5) 2004]

Implementation correctness via program checking

instead of verifying a transformation, verify its result.



Program checking goes back to Blum/Kannan (*"Programs that check their work"*)

[Glesner,Förster,Jäger: A Program Result Checker for the Lexical Analysis of the GNU C Compiler. COCV 2004.]

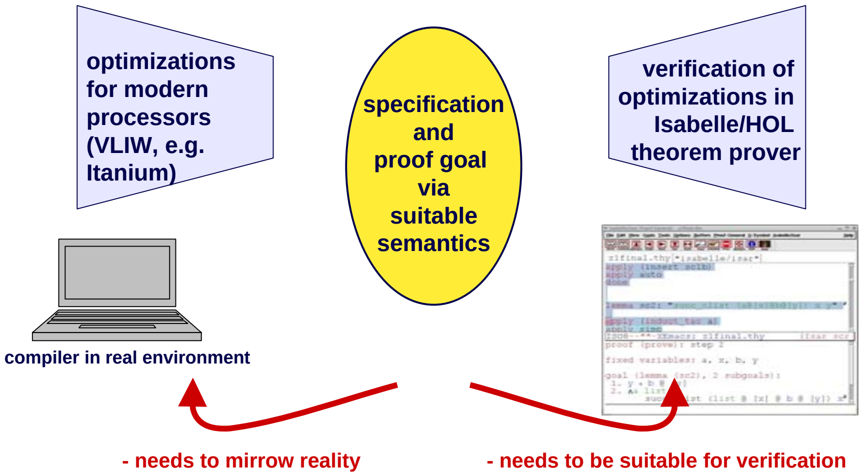
Verification of compilers

- absolutely essential for software development
- relatively large software systems
- semantically interesting
- results carry over to other software (and hardware) areas

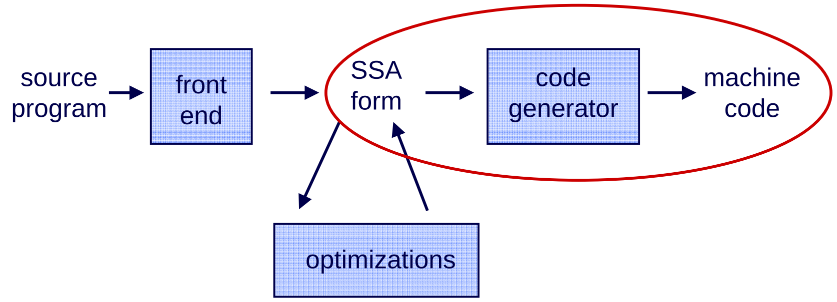
Overview

- Reasons and prerequisites for formal verification
- Verification of program / system transformations
- Three formal verifications as example:
  - Verification of optimizing compilers
  - Verification of model transformations
  - VATES: Verification of satellite software
- Overview about further research projects
- Conclusions and perspectives

Optimization and Verification in a Unifying Setting



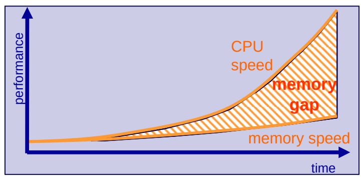
Tasks in compiler backends



SSA: static single assignment

Compiler Optimizations for VLIW Processors

- VLIW (*very long instruction word*):
  - potential to increase parallelism on instruction level
  - but current compilers do not use it
- Example Intel Itanium:
  - up to six instructions in parallel are possible
  - on average only three instructions executed in parallel
- Reasons:
  - memory gap: latency of up to 200 cycles for memory accesses
  - performance often dominated by memory speed (instead of CPU)
  - parallelism restricted by conservative analyses with imprecise results
    - „points-to"-sets of memory references: statically 23, dynamically only 1.06



⇒ Goal:
more precise analyses and speculative optimizations to overcome memory gap

Speculation: Implementation

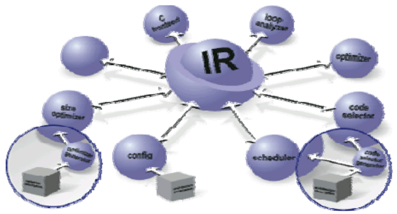
- speculative instructions may not change program semantics
- without hardware support:
  - only instructions without side effects speculatively
- with hardware support:
  - delay exceptions
  - run time tests much simpler

➔ hardware support not mandatory but useful
➔ Intel Itanium offers it

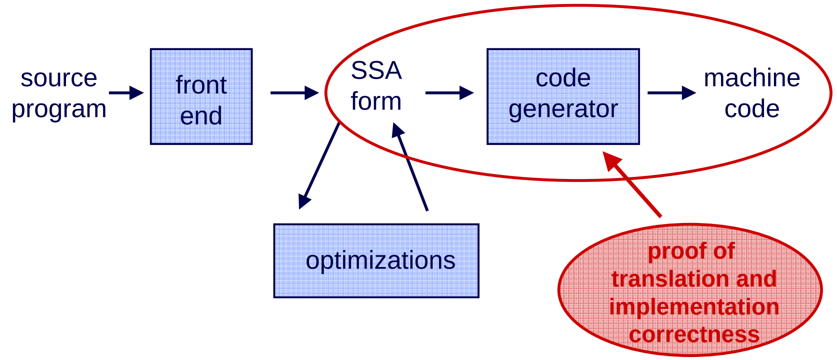
Compiler Optimizations – Platform

Cooperation with ACE (*Associated Compiler Experts*):

- CoSy-System for developing new optimizations
- Current state of the Itanium compiler:
  - nearly complete (i.e. nearly all Spec benchmarks run)
  - speculation prototypically integrated



Tasks in compiler backends



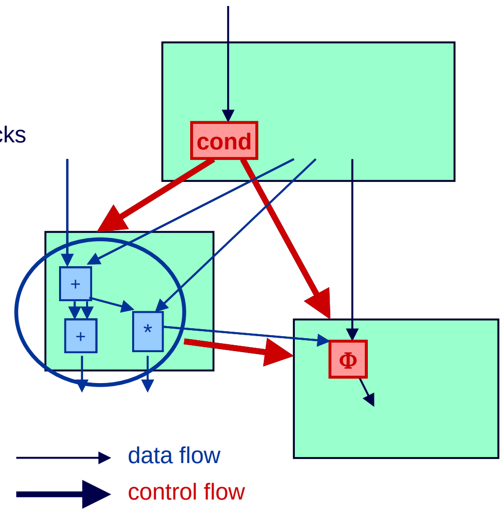
SSA: static single assignment

Formal semantics for SSA

Two layers:
- data flow in basic blocks
- control flow connecting basic blocks

In basic blocks:
- functional dependencies
- acyclic data flow graphs

Between basic blocks:
- imperative control flow
- state: active block + predecessor



Results translation correctness

- **specification of SSA basic blocks:**
  - as partial order
  - code generation creates additional dependencies (⇒ machine order)
- **machine proof for transformation between**
  - data-flow driven computation and
  - sequential instructions
- **correct iff data-flow dependencies are retained**
  - correct if SSA order ⊆ machine order
- **proof statistics:**
  - nearly 900 lopc (lines of proof code)
  - **proof can be reused (general proof princple)**

General Principle

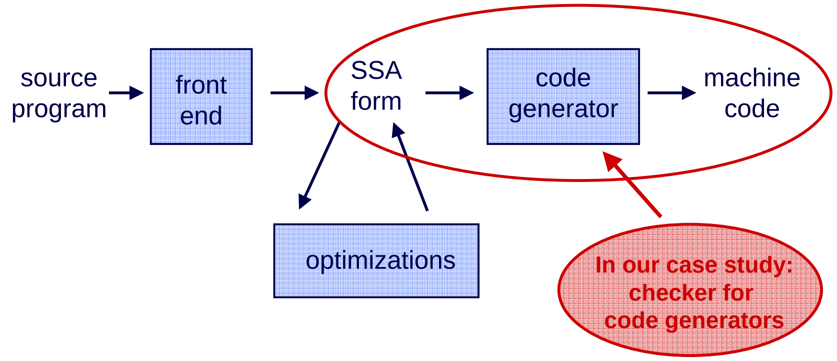
[Glesner, ASM 2004 + Blech,Glesner,ATPS 2004 + Blech,Glesner,Leitner,Mülling,COCV 2005]

Formalization of control flow

- based on operational semantics
- formalization as state transition sequences
- state transition sequences in theorem prover:
  - inductively defined as finite list
    - problem: non-terminating runs
  - coinductively defined as lazy lists
    - can also model non-terminating behavior
    - problem: Isabelle/HOL (as well as other theorem provers) coinductively not powerful enough
  - as bisimulations
    - relations that represent state transition behavior
    - can model non-terminating runs
    - can be represented adequately in Isabelle/HOL

[Glesner, COCV'04 & Leitner, Glesner, Blech, COCV 2006]

Tasks in compiler backends

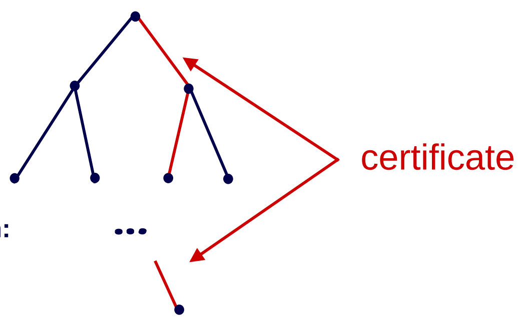


SSA: static single assignment

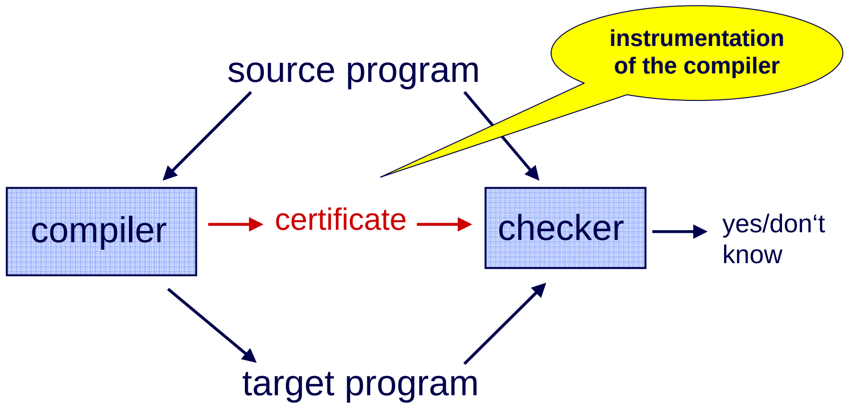
Program Checking for Optimizations

Problems in NP:
defined by proofs of polynomial length

Quality of solution:
not relevant for
correctness



Program Checking with Certificates



Backend Checking: Results

- **Compiler = non-deterministic Turing machine:**
  - searches for solution
  - computes solution
- **Checker = deterministic Turing machine:**
  - computes solution
- **Expectation:** checker code is part of compiler code

	code generator	checker
loc in .h-files	949	789
loc in .c-files	20887	10572
total loc	21836	11361

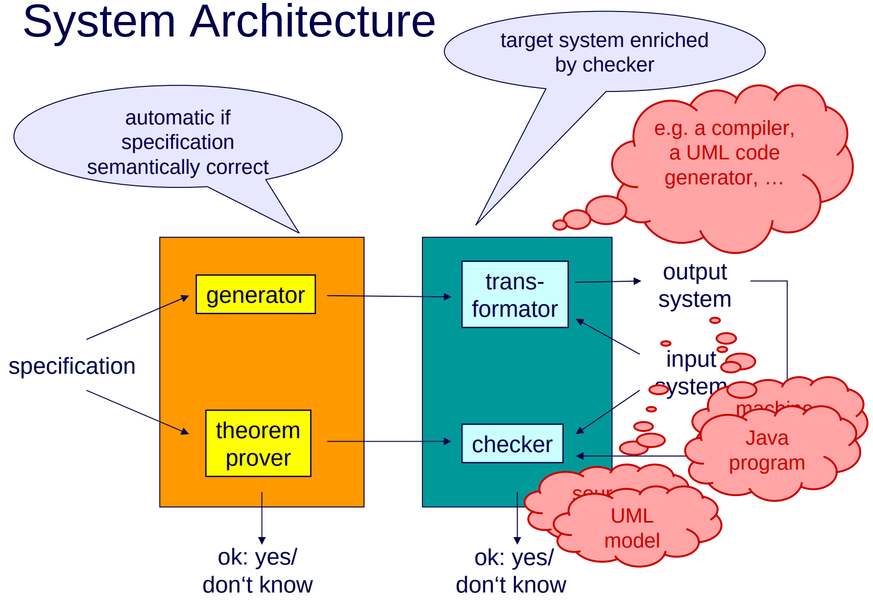
(Code generator from AJACS-Project with industrial partners)

Result:
Checker code identical with part of code generator

Consequence:
Substantial reduction of verification costs

[Glesner, J.UCS 2003 + Glesner, FME 2003]

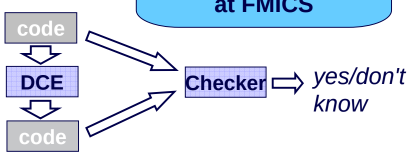
System Architecture



An Example for this Scenario: Verification of Dead Code Elimination

- Verification considers correctness of algorithm and correctness of implementation
- Correctness of algorithm verified within Isabelle/HOL
  - Formal Semantics for Static Single Assignment (SSA) Form
  - Formalization of Dead Code Algorithm
- Correctness of Implementation
  - Checker approach
  - Implemented as CoSy Engine

Further Verification Scenario presented at FMICS



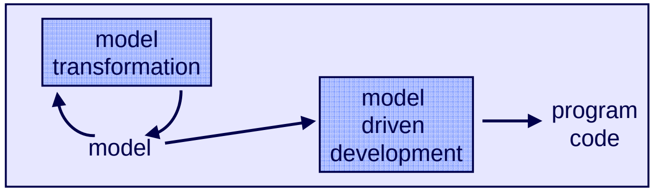
[Blech,Gesellensetter,Glesner; SEFM'05]

Overview

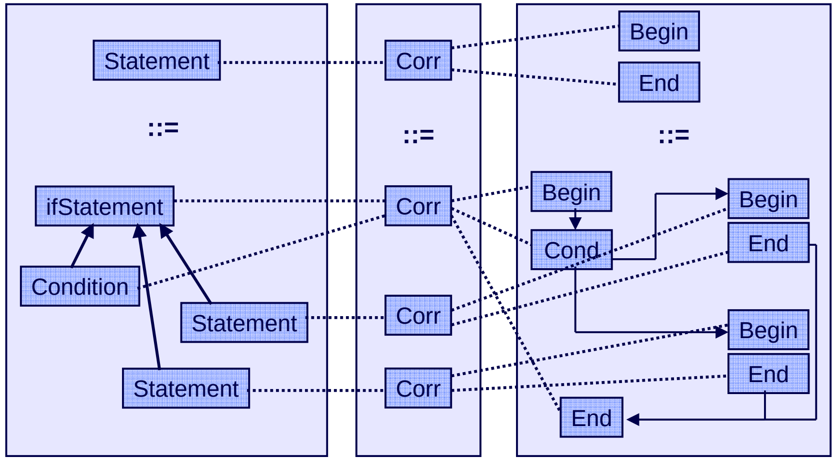
- Reasons and prerequisites for formal verification
- Verification of program / system transformations
- Three formal verifications as example:
  - Verification of optimizing compilers
  - Verification of model transformations
  - VATES: Verification of satellite software
- Overview about further research projects
- Conclusions and perspectives

Model transformations

- Model Driven Architecture (MDA) der Object Management Group (OMG)
  - model not only for documentation but also for development process
- models often represented by graphs
  - as relations between different objectes, etc.
  - see Unified Modeling Language (UML) as example
- specify transformation on models or transformations from models to code by graph transformation rules
  - Fujaba (**F**rom **U**ML to **J**ava **a**nd **b**ack **a**gain) tool suite at Paderborn



Example: simple TGG rule



Verification of TGG transformations

- To show:
  pairs of models as well as pairs of models and their corresponding programs are semantically equivalent
- verify semantic equivalence of all possible pairs inductively
  - axiom that models and programs, resp., in simple starting pair are equivalent
  - induction step that TGG rules preserve semantic equivalence
- can be expressed in Isabelle/HOL by inductive data types
- has been verified in Isabelle/HOL for basic transformations

[Blech, Glesner, Leitner, Fujaba Days 2005 +
Giese, Glesner, Leitner, Schäfer, Wagner, MoDeVa 2006]

Overview

- Reasons and prerequisites for formal verification
- Verification of program / system transformations
- Three formal verifications as example:
  - Verification of optimizing compilers
  - Verification of model transformations
  - VATES: Verification of satellite software
- Overview about further research projects
- Conclusions and perspectives

VATES

Deutsche Forschungsgemeinschaft DFG

- new project: **VATES**
  Verification **a**nd **T**ransformation of **E**mbedded **S**ystems
- construct and verify embedded, reactive, and concurrent systems
- verification throughout the whole process, from specification to machine code
- application: verification of BOSS, a RTOS used in practice for the BiRD satellite

Overview

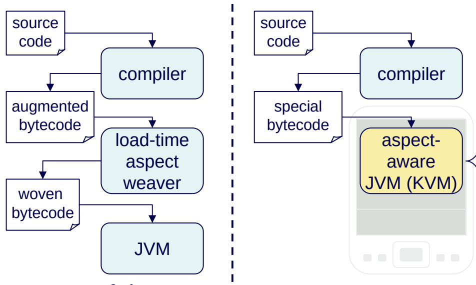
- Reasons and prerequisites for formal verification
- Verification of program / system transformations
- Three formal verifications as example:
  - Verification of optimizing compilers
  - Verification of model transformations
  - VATES: Verification of satellite software
- Overview about further research projects
- Conclusions and perspectives

Optimizations for Aspect-Oriented Programming Languages

- Aspect-orientation:
  - extension of the object-oriented paradigm
  - new modularization concept for system requirements that are orthogonal to the usual module structure
  - e.g. logging, synchronization, …
- In embedded systems:
  - variability at run time
  - dynamic extension and removal of aspects
  - necessary: optimization of the execution model

Optimized Aspect-Oriented Programming Languages

- for use in mobile embedded systems (low resources)
- concepts for optimizations to
- case study: dynamically adaptable mobile application



state-of-the-art

future work

- build-in aspect dispatch
  - enhanced data structures
  - additional instructions
- dynamic aspect activation (API)
- goal: efficient, dynamically adaptable program execution

HW/SW Co-Design

= integrated design of hardware and software
parts of embedded systems

Goals:

- early analysis of HW/SW-borders
- simplified system integration
- simulation, testing, verification
- evaluation of design alternatives

Members of my research group

??

??

T. Göthel (ab Aug.'07)

Dipl.-Ing. M. Beyer

Dipl.-Inform. C. Hundt

Dipl.-Inform. L. Gesellensetter

Dipl.-Inform. E. Salecker

Dipl.-Ing. P. Herber

… and many students doing projects and master theses with us

Conclusions and Perspectives

- **Verification and system construction/maintenance:**
  - best if from one source
  - together with optimizations (cf. compilers as example)
- **Formal verification:**
  - possible for large systems
  - generates engineering knowledge concerning
    - design, construction and maintenance

    for reliable software systems
- **Future application areas:**
  - software engineering
  - model transformations
  - hardware/software co-design
  - embedded systems
  - in general: safe and efficient systems

Thank you!

More information at:

pes.cs.tu-berlin.de