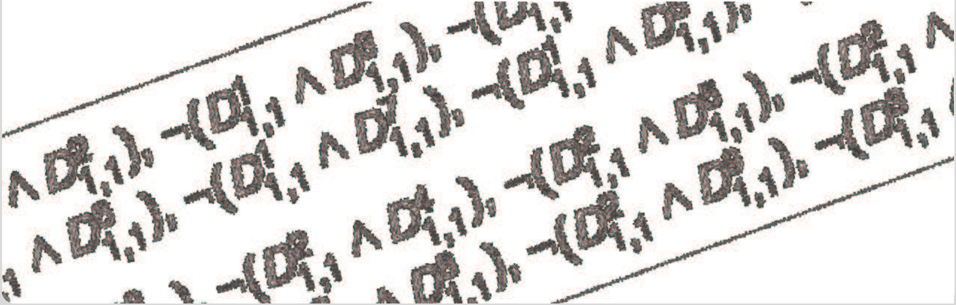
Karlsruhe Institute of Technology

Formal Verification of System Software

Experiences from the Verisoft XT Project

Bernhard Beckert | SVARM, 21.07.10

INSTITUTE FOR THEORETICAL INFORMATICS



The Karlsruhe Institute of Technology

Karlsruhe Institute of Technology

Merger of

- Karlsruhe University (state funded)
- Research Center Karlsruhe (funded by federal government)

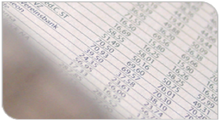
Figures

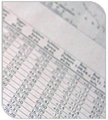
8,500 Employees

19,700 Students

364 Professors

650 Annual Budget in Million Euros





COST Action IC0701

Formal Verification of Object-Oriented Software

What is the Action About?

Formal Verification of Object-Oriented Software

Methods for . . .

- specification
- proving correctness

Tools to . . .

- automate the verification process

Integration of . . .

- specification and verification into mainstream software development tools and processes

What is the Action About?

Formal Verification of Object-Oriented Software

Methods for . . .

- specification
- proving correctness

Tools to . . .

- automate the verification process

Integration of . . .

- specification and verification into mainstream software development tools and processes

What is the Action About?

Formal Verification of Object-Oriented Software

Methods for . . .

- specification
- proving correctness

Tools to . . .

- automate the verification process

Integration of . . .

- specification and verification into mainstream software development tools and processes

What is the Action About?

Formal Verification of Object-Oriented Software

Methods for . . .

- specification
- proving correctness

Tools to . . .

- automate the verification process

Integration of . . .

- specification and verification into mainstream software development tools and processes

What is the Action About?

Formal Verification of Object-Oriented Software

Methods for . . .

- specification
- proving correctness

Tools to . . .

- automate the verification process

Integration of . . .

- specification and verification into mainstream software development tools and processes

What is the Action About?

Formal Verification of Object-Oriented Software

Methods for . . .

- specification
- proving correctness

Tools to . . .

- automate the verification process

Integration of . . .

- specification and verification into mainstream software development tools and processes

What is the Action About?

Formal Verification of Object-Oriented Software

Methods for . . .

- specification
- proving correctness

Tools to . . .

- automate the verification process

Integration of . . .

- specification and verification into mainstream software development tools and processes

Main Objective of the Action

To co-ordinate the research into verification technology

to achieve reach and power

needed to assure reliability of object-oriented programs

on industrial scale.

Secondary Objectives

Secondary Objectives of the Action

- Development and Standardisation of Specification Languages and Methods
- Standardisation of Tool Interfaces, Common Framework
- Education of Users in the Application of Tools and Methods
- Co-ordination of European Research in the Field
- Increase in Market Penetration of Formal Verification

Secondary Objectives

Secondary Objectives of the Action

- Development and Standardisation of Specification Languages and Methods
- Standardisation of Tool Interfaces, Common Framework
- Education of Users in the Application of Tools and Methods
- Co-ordination of European Research in the Field
- Increase in Market Penetration of Formal Verification

Secondary Objectives

Secondary Objectives of the Action

- Development and Standardisation of Specification Languages and Methods
- Standardisation of Tool Interfaces, Common Framework
- Education of Users in the Application of Tools and Methods
- Co-ordination of European Research in the Field
- Increase in Market Penetration of Formal Verification

Secondary Objectives

Secondary Objectives of the Action

- Development and Standardisation of Specification Languages and Methods
- Standardisation of Tool Interfaces, Common Framework
- Education of Users in the Application of Tools and Methods
- Co-ordination of European Research in the Field
- Increase in Market Penetration of Formal Verification

Secondary Objectives

Secondary Objectives of the Action

- Development and Standardisation of Specification Languages and Methods
- Standardisation of Tool Interfaces, Common Framework
- Education of Users in the Application of Tools and Methods
- Co-ordination of European Research in the Field
- Increase in Market Penetration of Formal Verification

Working Groups

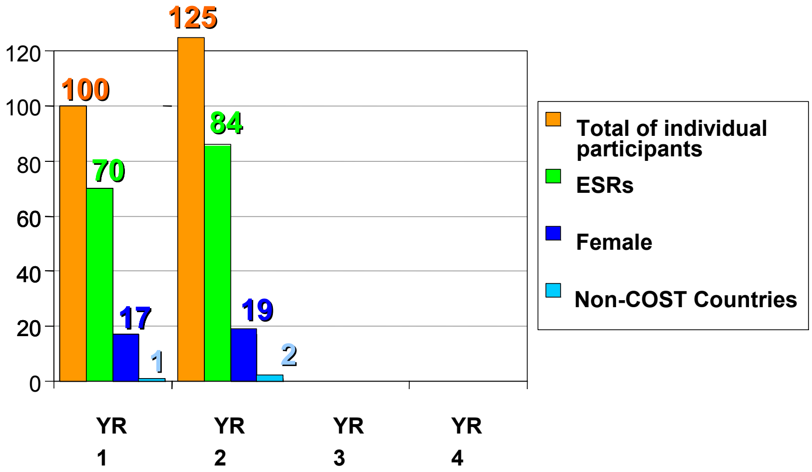
WG1: Customisable and Reusable Programs

WG2: Modularisation and Components

WG3: Concurrency

WG4: Tool Integration

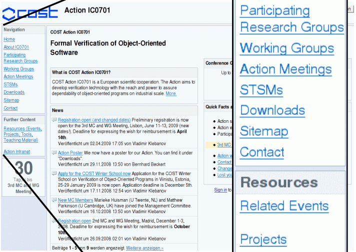
Action Participants

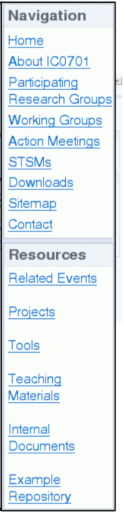


Action Website: www.cost-ic0701.org

Collection of Material

- inventories of tools and methods, projects, and events;
- teaching material;
- slides of all presentations given at meetings;





Collection of Verification Benchmarks

VerifyThis

- Web-based
- Examples from many different sources
- Independent of tools and specification languages

Topic of this Talk:

Deductive Program Verification for System Software

– Verisoft Project –

System Software

Mikrokernels

- Provide concurrency
- (Para-)virtualization
- System calls (e.g., for communication)
- Interface to interrupts and devices

Typical Properties

- Close to the hardware (assembly code)
- Platform-dependent

System Software

Mikrokernels

- Provide concurrency
- (Para-)virtualization
- System calls (e.g., for communication)
- Interface to interrupts and devices

Typical Properties

- Close to the hardware (assembly code)
- Platform-dependent

System Software

Requirements

- Functional correctness
- Security features (e.g., process separation)
- Fairness & liveness
- Realtime requirements

Verisoft Project

Phase I: 2003–2007

Verisoft Project, Phase I

KIT Karlsruhe Institute of Technology

Bundesministerium für Bildung und Forschung

AbsInt Angewandte Informatik

Audi

BOSCH

Concept of Truth

DFKI

Microsoft Innovation Center Europe

ESG

infineon

Microsoft Research

OneSpin Solutions

Sirrix AG security technologies

SYSGO

TÜV SÜD

Technische Universität Kaiserslautern

Technische Universität München

Universität Bremen

Universität des Saarlandes

Albert-Ludwigs-Universität Freiburg

Universität Koblenz · Landau

Verisoft Project, Phase I

- Complete (pervasive) formal verification of integrated computer system
- “Verifications as an Engineering Science”

Verisoft Project, Phase I

- Complete (pervasive) formal verification of integrated computer system
- "Verifications as an Engineering Science"

Four application areas (three from industry)

- Mobile phones (system-on-a-chip)
- Automotive
- Biometric identification
- Academic system

Verisoft Project, Phase I

- Complete (pervasive) formal verification of integrated computer system
- “Verifications as an Engineering Science”

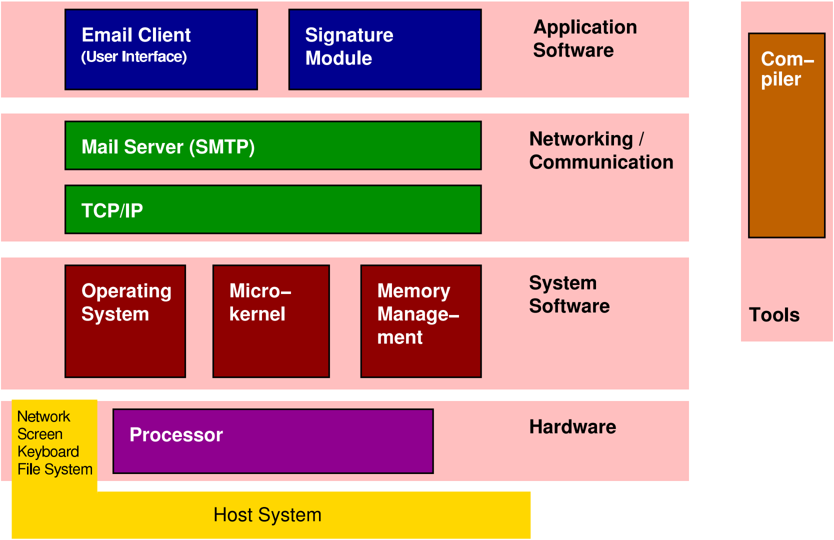
Four application areas (three from industry)

- Mobile phones (system-on-a-chip)
- Automotive
- Biometric identification
- Academic system

Funding

- Bundesministerium für Bildung und Forschung (bmb+f)
  approx. 4 Mio Euro / year
- Plus matching funds from industry

A Fully Verified Computer System



A Fully Verified Computer System

Implementation languages

- Gate-level description of processor
- DLX assembler
- C0

Verification technology

- Interactive verification with Isabelle

Verified properties

- Functional correctness
- Security properties
  - No access to private memory of other processes
  - Secure inter-process communication

A Fully Verified Computer System

Implementation languages

- Gate-level description of processor
- DLX assembler
- C0

Verification technology

- Interactive verification with Isabelle

Verified properties

- Functional correctness
- Security properties
  - No access to private memory of other processes
  - Secure inter-process communication

A Fully Verified Computer System

Implementation languages

- Gate-level description of processor
- DLX assembler
- C0

Verification technology

- Interactive verification with Isabelle

Verified properties

- Functional correctness
- Security properties
  - No access to private memory of other processes
  - Secure inter-process communication

Verisoft XT

Phase II: 2007–2010

Verisoft XT (Phase II)

Three (new) industrial applications

Hypervisor	Microsoft's Hypervisor (Kernel of Hyper-V) shipped with Microsoft Windows Server 2008
Avionics	Sysgo's PikeOS Microkernel Hypervisor for embedded systems
Automotive	Micro controler for safety-critical application in new Audi (timing analysis, model checking)

Verification technology

- Verification with VCC from Microsoft Research

Verisoft XT (Phase II)

Three (new) industrial applications

Hypervisor	Microsoft's Hypervisor (Kernel of Hyper-V) shipped with Microsoft Windows Server 2008
Avionics	Sysgo's PikeOS Microkernel Hypervisor for embedded systems
Automotive	Micro controler for safety-critical application in new Audi (timing analysis, model checking)

Verification technology

- Verification with VCC from Microsoft Research

Verisoft XT (Phase II)

Three (new) industrial applications

Hypervisor	Microsoft's Hypervisor (Kernel of Hyper-V) shipped with Microsoft Windows Server 2008
Avionics	Sysgo's PikeOS Microkernel Hypervisor for embedded systems
Automotive	Micro controler for safety-critical application in new Audi (timing analysis, model checking)

Verification technology

- Verification with VCC from Microsoft Research

Verisoft XT (Phase II)

Three (new) industrial applications

Hypervisor	Microsoft's Hypervisor (Kernel of Hyper-V) shipped with Microsoft Windows Server 2008
Avionics	Sysgo's PikeOS Microkernel Hypervisor for embedded systems
Automotive	Micro controler for safety-critical application in new Audi (timing analysis, model checking)

Verification technology

- Verification with VCC from Microsoft Research

Verisoft XT (Phase II)

Three (new) industrial applications

Hypervisor	Microsoft's Hypervisor (Kernel of Hyper-V) shipped with Microsoft Windows Server 2008
Avionics	Sysgo's PikeOS Microkernel Hypervisor for embedded systems
Automotive	Micro controler for safety-critical application in new Audi (timing analysis, model checking)

Verification technology

- Verification with VCC from Microsoft Research

Verifying the PikeOS Microkernel

Avionics Application (Sysgo's PikeOS)

- L4-based, industrial, safety- and security-critical microkernel
- Flies in Airbus A350
- ca. 20,000 LOC (90% C, 10% assembler)

Challenges

- Complexity and size of kernel
- "Real" implementation in C + assembly
- Not implemented with verification in mind
- Concurrency (with preemption)

Verifying the PikeOS Microkernel

Avionics Application (Sysgo's PikeOS)

- L4-based, industrial, safety- and security-critical microkernel
- Flies in Airbus A350
- ca. 20,000 LOC (90% C, 10% assembler)

Challenges

- Complexity and size of kernel
- "Real" implementation in C + assembly
- Not implemented with verification in mind
- Concurrency (with preemption)

Verifying the PikeOS Microkernel

Avionics Application (Sysgo's PikeOS)

- L4-based, industrial, safety- and security-critical microkernel
- Flies in Airbus A350
- ca. 20,000 LOC (90% C, 10% assembler)

Challenges

- Complexity and size of kernel
- "Real" implementation in C + assembly
- Not implemented with verification in mind
- Concurrency (with preemption)

Verifying the PikeOS Microkernel

Avionics Application (Sysgo's PikeOS)

- L4-based, industrial, safety- and security-critical microkernel
- Flies in Airbus A350
- ca. 20,000 LOC (90% C, 10% assembler)

Challenges

- Complexity and size of kernel
- "Real" implementation in C + assembly
- Not implemented with verification in mind
- Concurrency (with preemption)

Verifying the PikeOS Microkernel

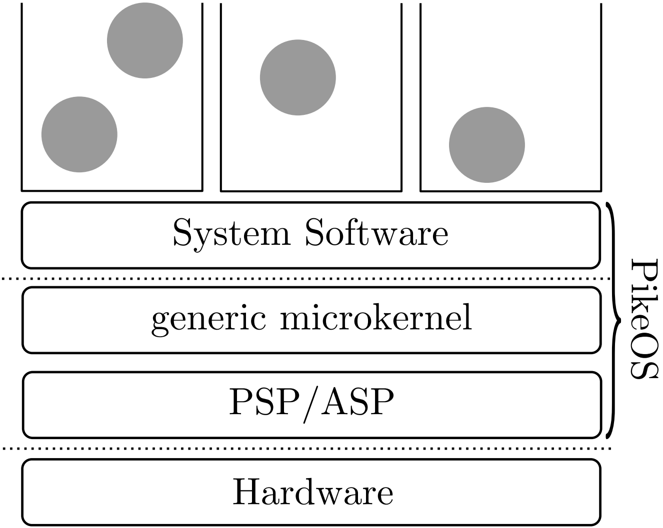
Avionics Application (Sysgo's PikeOS)

- L4-based, industrial, safety- and security-critical microkernel
- Flies in Airbus A350
- ca. 20,000 LOC (90% C, 10% assembler)

Challenges

- Complexity and size of kernel
- "Real" implementation in C + assembly
- Not implemented with verification in mind
- Concurrency (with preemption)

Verifying the PikeOS Microkernel



Verifying the PikeOS Microkernel

Karlsruhe Institute of Technology

Implementation properties and context

- Single-processor system (limited concurrency)
- *Para*-virtualising
  (more a microkernel than a hypervisor)
- PikeOS system software has to be taken into consideration
- Preemption

Verifying the PikeOS Microkernel

Implementation properties and context

- Single-processor system (limited concurrency)
- *Para*-virtualising
  (more a microkernel than a hypervisor)
- PikeOS system software has to be taken into consideration
- Preemption

Verifying the PikeOS Microkernel

Implementation properties and context

- Single-processor system (limited concurrency)
- *Para*-virtualising
  (more a microkernel than a hypervisor)
- PikeOS system software has to be taken into consideration
- Preemption

Verifying the PikeOS Microkernel

Implementation properties and context

- Single-processor system (limited concurrency)
- *Para*-virtualising
  (more a microkernel than a hypervisor)
- PikeOS system software has to be taken into consideration
- Preemption

Microsoft's VCC

Approach: Verifying Compiler

- Specification annotated directly in the program text (pre-/post-conditions, invariants, ownership, ...)
- Generate verification conditions, give them to SMT solver (tool chain)
- Automatic proof construction (interaction by changing the input)
- Concurency: Rely-guarantee paradigm

Microsoft's VCC

Approach: Verifying Compiler

- Specification annotated directly in the program text (pre-/post-conditions, invariants, ownership, . . .)
- Generate verification conditions, give them to SMT solver (tool chain)
- Automatic proof construction (interaction by changing the input)
- Concurency: Rely-guarantee paradigm

Microsoft's VCC

Approach: Verifying Compiler

- Specification annotated directly in the program text (pre-/post-conditions, invariants, ownership, . . .)
- Generate verification conditions, give them to SMT solver (tool chain)
- Automatic proof construction (interaction by changing the input)
- Concurency: Rely-guarantee paradigm

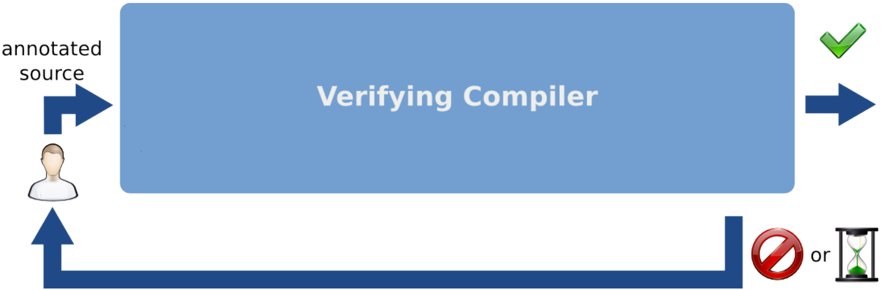
Microsoft's VCC

Karlsruhe Institute of Technology

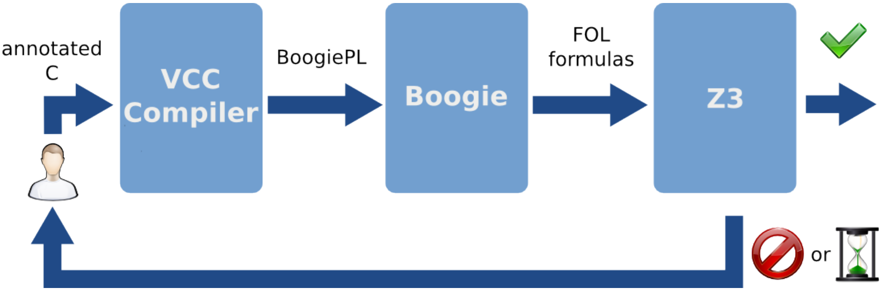
Approach: Verifying Compiler

- Specification annotated directly in the program text (pre-/post-conditions, invariants, ownership, . . .)
- Generate verification conditions, give them to SMT solver (tool chain)
- Automatic proof construction (interaction by changing the input)
- Concurency: Rely-guarantee paradigm

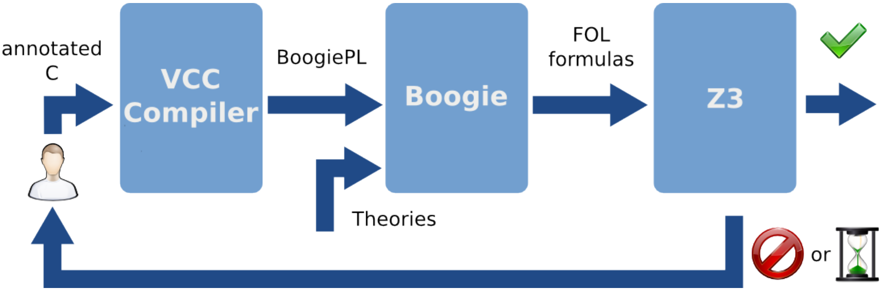
Microsoft's VCC



The VCC Toolchain



The VCC Toolchain

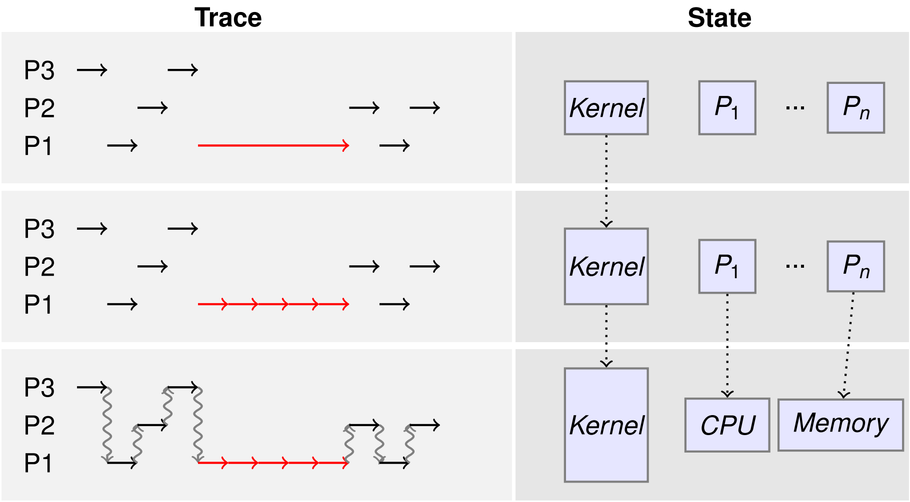


Example Specification (Sequential)

```
P4_prio_t p4_runner_changeprio
(P4k_thrinfo_t *proc, P4_prio_t newprio)
  requires(proc ==
              abstractModel.currentThread)
  ensures(proc->schedprio == newprio && ...)
  returns(old(proc->userprio))

  maintains(wrapped(...))
  writes(...)
{
 ...
}
```

Structure of the Overall Proof



State and Achievements of the Project

Methodological

- Method for handling memory management (allocation, page handling, separation, . . .)
- Method for handling inline assembler
- Method for handling preemption

Abstract model

- Abstract model of the kernel (tasks, threads, IPC-related, . . .)

Verification

- Partial verification of the code – sequential and concurrent (complete verification possible but for lack in man-power)

State and Achievements of the Project

Methodological

- Method for handling memory management
  (allocation, page handling, separation, . . .)
- Method for handling inline assembler
- Method for handling preemption

Abstract model

- Abstract model of the kernel
  (tasks, threads, IPC-related, . . .)

Verification

- Partial verification of the code – sequential and concurrent
  (complete verification possible but for lack in man-power)

State and Achievements of the Project

Methodological

- Method for handling memory management (allocation, page handling, separation, . . .)
- Method for handling inline assembler
- Method for handling preemption

Abstract model

- Abstract model of the kernel (tasks, threads, IPC-related, . . .)

Verification

- Partial verification of the code – sequential and concurrent (complete verification possible but for lack in man-power)

State and Achievements of the Project

Methodological

- Method for handling memory management
  (allocation, page handling, separation, ...)
- Method for handling inline assembler
- Method for handling preemption

Abstract model

- Abstract model of the kernel
  (tasks, threads, IPC-related, ...)

Verification

- Partial verification of the code – sequential and concurrent
  (complete verification possible but for lack in man-power)

State and Achievements of the Project

Methodological

- Method for handling memory management (allocation, page handling, separation, . . .)
- Method for handling inline assembler
- Method for handling preemption

Abstract model

- Abstract model of the kernel (tasks, threads, IPC-related, . . .)

Verification

- Partial verification of the code – sequential and concurrent (complete verification possible but for lack in man-power)

State and Achievements of the Project

Methodological

- Method for handling memory management
  (allocation, page handling, separation, . . .)
- Method for handling inline assembler
- Method for handling preemption

Abstract model

- Abstract model of the kernel
  (tasks, threads, IPC-related, . . .)

Verification

- Partial verification of the code – sequential and concurrent
  (complete verification possible but for lack in man-power)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand $\longrightarrow$ specify $\longrightarrow$ verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand → specify → verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand $\longrightarrow$ specify $\longrightarrow$ verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand → specify → verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand $\longrightarrow$ specify $\longrightarrow$ verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand $\longrightarrow$ specify $\longrightarrow$ verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand $\longrightarrow$ specify $\longrightarrow$ verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand → specify → verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand $\longrightarrow$ specify $\longrightarrow$ verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Bottom-up vs. top-down

- Bottom-up: start with individual functions
- Top-down: start with global requirement spec

We started bottom-up

- understand $\longrightarrow$ specify $\longrightarrow$ verify (iterative)
  - helper functions
  - individual system calls
- first sequential, then concurrent

Result

Verification of functions / system calls possible with VCC
(both sequential and concurrent)

Insights

Top-down understanding / specification

- Understanding of global data structures, scheduling mechanism, etc. more difficult to achieve than an understanding of individual functions
- VCC is not ideal for *developing* top-level models:
  - local (single thread) focus, obfuscation of global view

Therefore . . .

Decided to use Isabelle for *developing* formal top-level model of the kernel

Insights

Karlsruhe Institute of Technology

Top-down understanding / specification

- Understanding of global data structures, scheduling mechanism, etc. more difficult to achieve than an understanding of individual functions
- VCC is not ideal for *developing* top-level models:
  - local (single thread) focus, obfuscation of global view

Therefore ...

Decided to use Isabelle for *developing* formal top-level model of the kernel

Insights

Top-down understanding / specification

- Understanding of global data structures, scheduling mechanism, etc. more difficult to achieve than an understanding of individual functions
- VCC is not ideal for *developing* top-level models:
  - local (single thread) focus, obfuscation of global view

Therefore . . .

Decided to use Isabelle for *developing* formal top-level model of the kernel

Insights

Top-down understanding / specification

- Understanding of global data structures, scheduling mechanism, etc. more difficult to achieve than an understanding of individual functions
- VCC is not ideal for *developing* top-level models:
  - local (single thread) focus, obfuscation of global view

Therefore ...

Decided to use Isabelle for *developing* formal top-level model of the kernel

Insights

Top-down understanding / specification

- Understanding of global data structures, scheduling mechanism, etc. more difficult to achieve than an understanding of individual functions
- VCC is not ideal for *developing* top-level models:
  - local (single thread) focus, obfuscation of global view

Therefore ...

Decided to use Isabelle for *developing* formal top-level model of the kernel

Handling Concurrency

Example: System Call p4_fast_set_prio

Karlsruhe Institute of Technology

From the kernel reference manual

"This function sets the current thread's priority to newprio. Invalid or too high priorities are limited to the caller's task MCP. Upon success, a call to this function returns the current thread's priority before setting it to newprio."

Implementation

```
P4_prio_t p4_runner_changeprio
(P4k_thrinfo_t *proc, P4_prio_t newprio)
{
  P4_prio_t oldprio; P4_cpureg_t oldstat;

  oldstat = p4arch_disable_int();
    oldprio = proc->userprio;
    proc->userprio = newprio;
    proc->schedprio = newprio;
    kglobal.kinfo->currprio = newprio;
  p4arch_restore_int(oldstat);

  return oldprio;
}
```

Specification (Sequential)

```
P4_prio_t p4_runner_changeprio
(P4k_thrinfo_t *proc, P4_prio_t newprio)
  requires(proc ==
           abstractModel.currentThread)
  ensures(proc->schedprio == newprio && ...)
  returns(old(proc->userprio))

  maintains(wrapped(...))
  writes(...)
{
 ...
}
```

Specification (Concurrent)

```
void setPrio(pThread t, int v
         spec(update *up))
    maintains(up->value == v)

    requires(set_in((obj_t) up, t->hist)
         && !set_in((obj_t) up, t->done))

    ensures(exists(update *u;
        set_in((obj_t) u, t->done)
        && !set_in((obj_t) u, old(t->done))
        && t->prio == u->value))

    ensures(set_in((obj_t) up, t->done))
```

Handling Preemption

Preemption

- Means of scheduling (context switch)
- Pause current C thread in kernel, run other
- Voluntarily (IPC) or forced (interrupts)

Separate verification tasks

- Verify thread switch to be correct (once)
- Verify individual system call using thread-switch properties as axioms

Handling Preemption

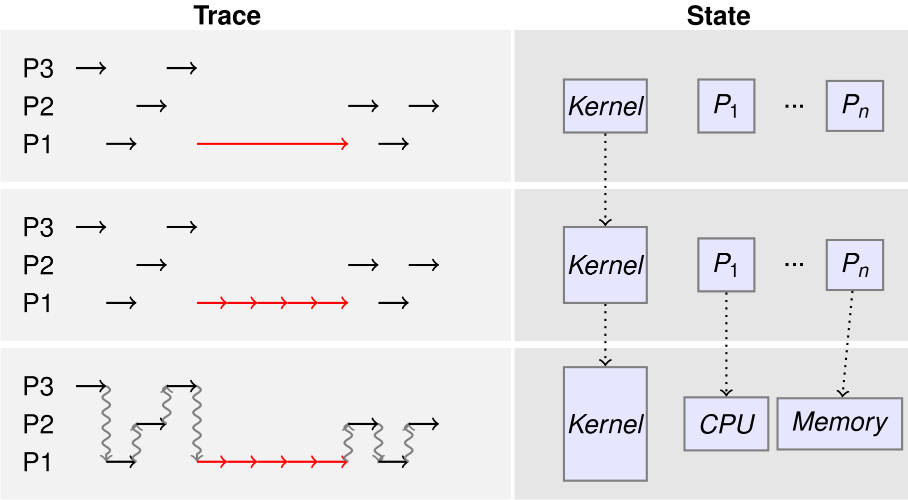
Preemption

- Means of scheduling (context switch)
- Pause current C thread in kernel, run other
- Voluntarily (IPC) or forced (interrupts)

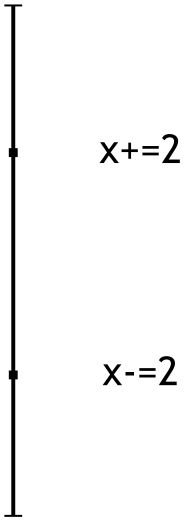
Separate verification tasks

- Verify thread switch to be correct (once)
- Verify individual system call using thread-switch properties as axioms

Structure of the Overall Proof



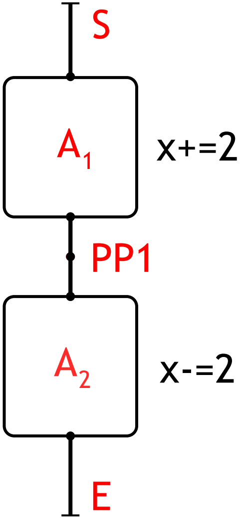
Running Example: Boostprio (abstract)



Tasks:

- identify atomic blocks in the kernel and preemption points
- specify effect of each atomic block as transition over abstract state
- introduce additional information about state; add axioms
- deduce intended top-level property

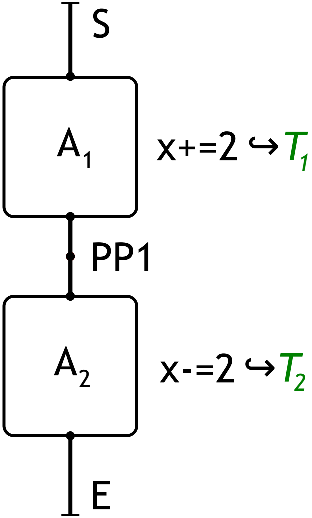
Running Example: Boostprio (abstract)



Tasks:

- identify atomic blocks in the kernel and preemption points
- specify effect of each atomic block as transition over abstract state
- introduce additional information about state; add axioms
- deduce intended top-level property

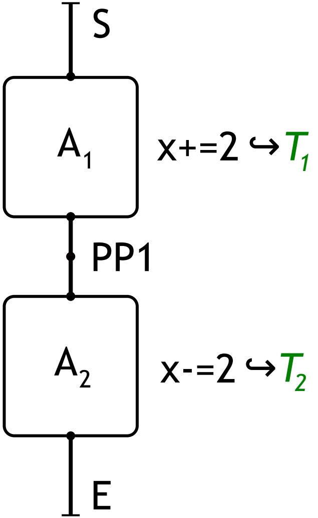
Running Example: Boostprio (abstract)



Tasks:

- identify atomic blocks in the kernel and preemption points
- specify effect of each atomic block as transition over abstract state
- introduce additional information about state; add axioms
- deduce intended top-level property

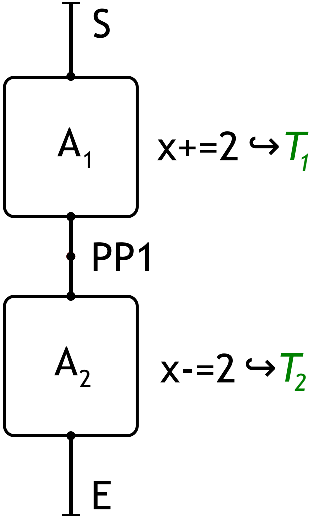
System Transitions



Specify effect of each atomic block as transition over abstract state:

T_1 : old(location) == S
$\wedge$ location == PP1
$\wedge$ x == old(x) + 2

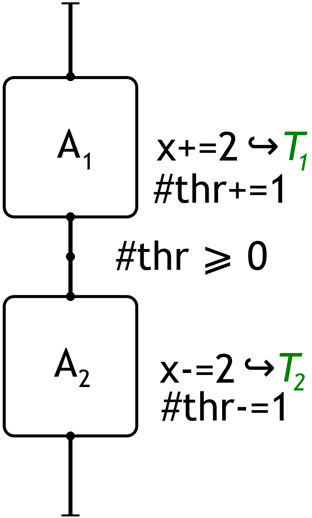
Running Example: Boostprio (abstract)



Tasks:

- identify atomic blocks in the kernel and preemption points
- specify effect of each atomic block as transition over abstract state
- introduce additional information about state; add axioms
- deduce intended top-level property

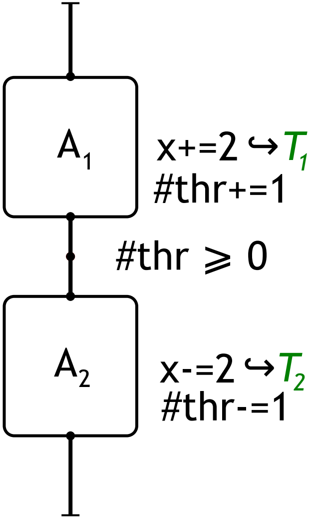
Running Example: Boostprio (abstract)



Tasks:

- identify atomic blocks in the kernel and preemption points
- specify effect of each atomic block as transition over abstract state
- introduce additional information about state; add axioms
- deduce intended top-level property

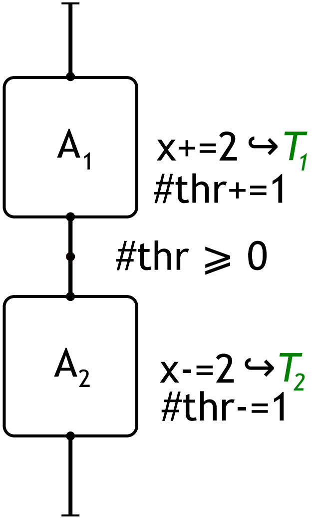
Syscall Specification



Introduce additional information about state:

- #thr represents number of threads at PP1
- ”bookkeeping“ by ghost code
- #thr >= 0 follows from scheduler properties

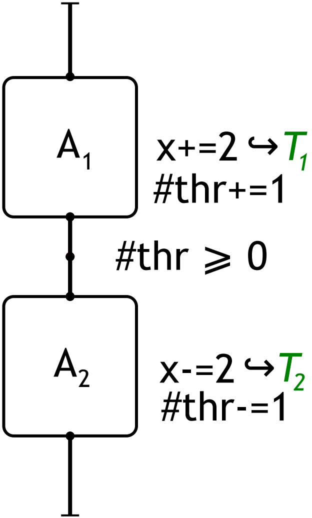
Running Example: Boostprio (abstract)



Tasks:

- identify atomic blocks in the kernel and preemption points
- specify effect of each atomic block as transition over abstract state
- introduce additional information about state; add axioms
- deduce intended top-level property

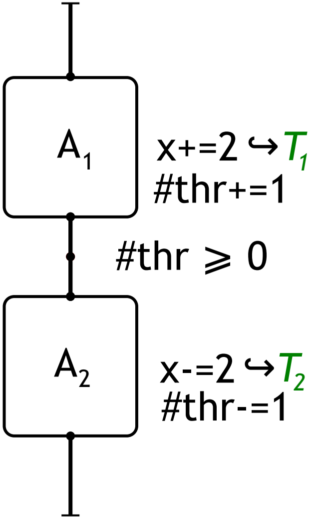
Running Example: Boostprio (abstract)



Tasks:

- identify atomic blocks in the kernel and preemption points
- specify effect of each atomic block as transition over abstract state
- introduce additional information about state; add axioms
- deduce intended top-level property

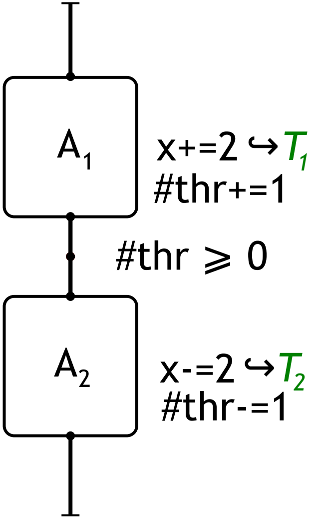
Running Example: Boostprio (abstract)



Tasks:

- identify atomic blocks in the kernel and preemption points
- specify effect of each atomic block as transition over abstract state
- introduce additional information about state; add axioms
- deduce intended top-level property

Syscall Specification



Deduce intended top-level property:

Given:

- Initial state: x == 0
- Axiom: #thr >= 0
- Invariant: x >= 2 * #thr

We can show our intended property:

x >= 0

Completeness of Verifying Compilers

Relative Completeness

Given

P: program

REQ: requirement specification

$\models$ definition of when a program satisfies a specification

$(\vdash_S, Th_S)$: verification system (deduction relation, axioms)

Relative Completeness

$(\vdash_S, Th_S)$ is *relatively complete* (w.r.t. arithmetics) if,
for each program P and specification REQ with

$$\models P{+}REQ\ ,$$

there is a set *Arith* of valid arithmetical formulas such that

$$Th_S \cup Arith \vdash_S P{+}REQ\ .$$

Relative Completeness

Given

P: program

REQ: requirement specification

$\models$ definition of when a program satisfies a specification

$(\vdash_S, Th_S)$: verification system (deduction relation, axioms)

Relative Completeness

$(\vdash_S, Th_S)$ is *relatively complete* (w.r.t. arithmetics) if,
for each program P and specification REQ with

$$\models P{+}REQ \ ,$$

there is a set *Arith* of valid arithmetical formulas such that

$$Th_S \cup Arith \vdash_S P{+}REQ \ .$$

Completeness: Theory vs. Practice

Karlsruhe Institute of Technology

Verifying Compiler in Theory

- There exist Verifying Compilers that are relatively complete
- For these, providing *P*+*REQ* is sufficient
- All auxiliary annotations can be effectively computed (e.g. loop invariants)
- But . . .
  - "easily" generated invariants use Gödelisaion
  - are useless in practice

VC in Practice

- Verifying Compilers are *not* relatively complete
- User has to provide *the right* auxiliary annotations

Completeness: Theory vs. Practice

Verifying Compiler in Theory

- There exist Verifying Compilers that are relatively complete
- For these, providing $P{+}REQ$ is sufficient
- All auxiliary annotations can be effectively computed (e.g. loop invariants)
- But …
  - "easily" generated invariants use Gödelisaion
  - are useless in practice

VC in Practice

- Verifying Compilers are *not* relatively complete
- User has to provide *the right* auxiliary annotations

Completeness: Theory vs. Practice

Verifying Compiler in Theory

- There exist Verifying Compilers that are relatively complete
- For these, providing $P+REQ$ is sufficient
- All auxiliary annotations can be effectively computed (e.g. loop invariants)
- But . . .
  - "easily" generated invariants use Gödelisaion
  - are useless in practice

VC in Practice

- Verifying Compilers are *not* relatively complete
- User has to provide *the right* auxiliary annotations

Completeness: Theory vs. Practice

Verifying Compiler in Theory

- There exist Verifying Compilers that are relatively complete
- For these, providing $P+REQ$ is sufficient
- All auxiliary annotations can be effectively computed (e.g. loop invariants)
- But . . .
  - "easily" generated invariants use Gödelisaion
  - are useless in practice

VC in Practice

- Verifying Compilers are *not* relatively complete
- User has to provide *the right* auxiliary annotations

Completeness: Theory vs. Practice

KIT Karlsruhe Institute of Technology

Verifying Compiler in Theory

- There exist Verifying Compilers that are relatively complete
- For these, providing *P*+*REQ* is sufficient
- All auxiliary annotations can be effectively computed (e.g. loop invariants)
- But . . .
  - “easily” generated invariants use Gödelisaion
  - are useless in practice

VC in Practice

- Verifying Compilers are *not* relatively complete
- User has to provide *the right* auxiliary annotations

Completeness: Theory vs. Practice

KIT Karlsruhe Institute of Technology

Verifying Compiler in Theory

- There exist Verifying Compilers that are relatively complete
- For these, providing $P{+}REQ$ is sufficient
- All auxiliary annotations can be effectively computed (e.g. loop invariants)
- But . . .
  - "easily" generated invariants use Gödelisaion
  - are useless in practice

VC in Practice

- Verifying Compilers are *not* relatively complete
- User has to provide *the right* auxiliary annotations

Completeness: Theory vs. Practice

Verifying Compiler in Theory

- There exist Verifying Compilers that are relatively complete
- For these, providing *P+REQ* is sufficient
- All auxiliary annotations can be effectively computed (e.g. loop invariants)
- But . . .
  - “easily” generated invariants use Gödelisaion
  - are useless in practice

VC in Practice

- Verifying Compilers are *not* relatively complete
- User has to provide *the right* auxiliary annotations

Completeness: Theory vs. Practice

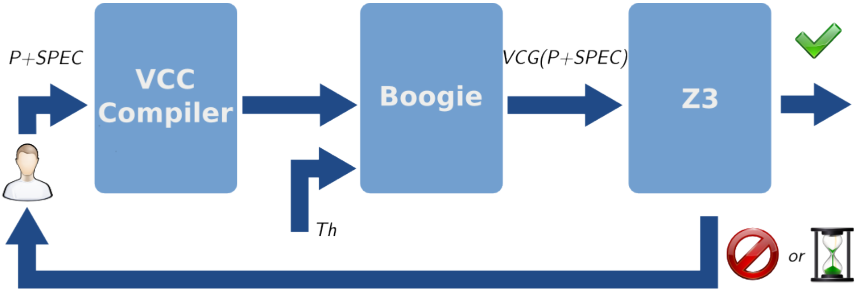
Verifying Compiler in Theory

- There exist Verifying Compilers that are relatively complete
- For these, providing $P{+}REQ$ is sufficient
- All auxiliary annotations can be effectively computed (e.g. loop invariants)
- But . . .
  - “easily” generated invariants use Gödelisaion
  - are useless in practice

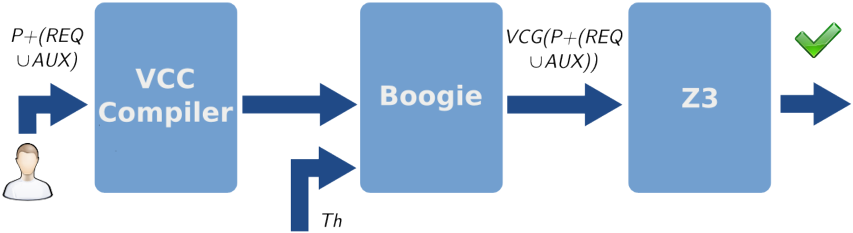
VC in Practice

- Verifying Compilers are *not* relatively complete
- User has to provide *the right* auxiliary annotations

The VCC Toolchain



Completeness of Verifying Compilers



Annotation Completeness

Annotation Completeness of Verifying Compilers

$(\vdash_S, Th_S)$ is *annotation complete* if,
for each program P and specification REQ with

$$\models P{+}REQ \quad ,$$

there is

- a set AUX of annotations,
- a set $Arith$ of valid arithmetical formulas

such that

$$Th_S \cup Arith \vdash_S P{+}(REQ \cup AUX) \quad .$$

Insights and Conclusions

Conclusions I: Success!

Success!

- Verification of microkernel
  – i.e., complex concurrent C code –
  possible with VCC
- Rely-guarantee is a successful approach for verifying concurrent C
- Modern SMT solvers are powerful

Conclusions I: Success!

Success!

- Verification of microkernel
  – i.e., complex concurrent C code –
  possible with VCC
- Rely-guarantee is a successful approach for verifying concurrent C
- Modern SMT solvers are powerful

Conclusions I: Success!

Karlsruhe Institute of Technology

Success!

- Verification of microkernel
  – i.e., complex concurrent C code –
  possible with VCC
- Rely-guarantee is a successful approach for verifying concurrent C
- Modern SMT solvers are powerful

Conclusions II: But . . .

Effort

- Specifying and verifying complex systems is still a huge effort
- In particular if the system is not built with verification in mind

VCC (and similar tools) . . .

- are *not* "push button"
- are not ideal for *developing* top-level models

Not a good idea . . .

- to a use tool while it is being developed for complex verification task

Conclusions II: But . . .

Effort

- Specifying and verifying complex systems is still a huge effort
- In particular if the system is not built with verification in mind

VCC (and similar tools) . . .

- are *not* “push button”
- are not ideal for *developing* top-level models

Not a good idea . . .

- to a use tool while it is being developed for complex verification task

Conclusions II: But . . .

Effort

- Specifying and verifying complex systems is still a huge effort
- In particular if the system is not built with verification in mind

VCC (and similar tools) . . .

- are *not* “push button”
- are not ideal for *developing* top-level models

Not a good idea . . .

- to a use tool while it is being developed for complex verification task

Conclusions II: But . . .

Effort

- Specifying and verifying complex systems is still a huge effort
- In particular if the system is not built with verification in mind

VCC (and similar tools) . . .

- are *not* “push button”
- are not ideal for *developing* top-level models

Not a good idea . . .

- to a use tool while it is being developed for complex verification task

Conclusions II: But . . .

Effort

- Specifying and verifying complex systems is still a huge effort
- In particular if the system is not built with verification in mind

VCC (and similar tools) . . .

- are *not* “push button”
- are not ideal for *developing* top-level models

Not a good idea . . .

- to a use tool while it is being developed for complex verification task

Conclusion III: What Users Need to Know about Internals of Tools

In Theory

- Users do not need knowledge about internals

In Practice

To provide useful annotations, users need…

- knowledge about how to influence proof search
- to be aware of the distinction between
  - requirement and auxiliary annotations
  - essential and non-essential annotations

Conclusion III: What Users Need to Know about Internals of Tools

In Theory

- Users do not need knowledge about internals

In Practice

To provide useful annotations, users need…

- knowledge about how to influence proof search
- to be aware of the distinction between
  - requirement and auxiliary annotations
  - essential and non-essential annotations

Conclusion III: What Users Need to Know about Internals of Tools

In Theory

- Users do not need knowledge about internals

In Practice

To provide useful annotations, users need...

- knowledge about how to influence proof search
- to be aware of the distinction between
  - requirement and auxiliary annotations
  - essential and non-essential annotations

Conclusion III: What Users Need to Know about Internals of Tools

In Theory

- Users do not need knowledge about internals

In Practice

To provide useful annotations, users need...

- knowledge about how to influence proof search
- to be aware of the distinction between
  - requirement and auxiliary annotations
  - essential and non-essential annotations

Conclusion III: What Users Need to Know about Internals of Tools

In Theory

- Users do not need knowledge about internals

In Practice

To provide useful annotations, users need...

- knowledge about how to influence proof search
- to be aware of the distinction between
  - requirement and auxiliary annotations
  - essential and non-essential annotations

Conclusion III: What Users Need to Know about Internals of Tools

In Theory

- Users do not need knowledge about internals

In Practice

To provide useful annotations, users need. . .

- knowledge about how to influence proof search
- to be aware of the distinction between
  - requirement and auxiliary annotations
  - essential and non-essential annotations

Requirement vs. Auxiliary Specification

Annotations serve different purposes

Requirement specification: Express the properties to be verified

Auxiliary annotations: Provide knowledge about the program
Support the verification process / finding a proof

Two kinds of auxiliary annotations

Needed for efficiency (lemmas):
Allows shorter and/or easier to find proofs
Not needed for the existence of a proof

Essential: Needed for the existence of a proof
loop invariants, assignable clauses

Requirement vs. Auxiliary Specification

Annotations serve different purposes

Requirement specification: Express the properties to be verified

Auxiliary annotations: Provide knowledge about the program
Support the verification process / finding a proof

Two kinds of auxiliary annotations

Needed for efficiency (lemmas):
Allows shorter and/or easier to find proofs
Not needed for the existence of a proof

Essential: Needed for the existence of a proof
loop invariants, assignable clauses

Requirement vs. Auxiliary Specification

Annotations serve different purposes

Requirement specification: Express the properties to be verified

Auxiliary annotations: Provide knowledge about the program
Support the verification process / finding a proof

Two kinds of auxiliary annotations

Needed for efficiency (lemmas):
Allows shorter and/or easier to find proofs
Not needed for the existence of a proof

Essential: Needed for the existence of a proof
loop invariants, assignable clauses

Requirement vs. Auxiliary Specification

Annotations serve different purposes

Requirement specification: Express the properties to be verified

Auxiliary annotations: Provide knowledge about the program
Support the verification process / finding a proof

Two kinds of auxiliary annotations

Needed for efficiency (lemmas):

- Allows shorter and/or easier to find proofs
- *Not needed* for the existence of a proof

Essential:

- *Needed* for the existence of a proof:
  loop invariants, assignable clauses, ...

Requirement vs. Auxiliary Specification

Annotations serve different purposes

Requirement specification: Express the properties to be verified

Auxiliary annotations: Provide knowledge about the program
Support the verification process / finding a proof

Two kinds of auxiliary annotations

Needed for efficiency (lemmas):

- Allows shorter and/or easier to find proofs
- *Not needed* for the existence of a proof

Essential:

- *Needed* for the existence of a proof:
  loop invariants, assignable clauses, ...

Requirement vs. Auxiliary Specification

Annotations serve different purposes

Requirement specification: Express the properties to be verified

Auxiliary annotations: Provide knowledge about the program
Support the verification process / finding a proof

Two kinds of auxiliary annotations

Needed for efficiency (lemmas):

- Allows shorter and/or easier to find proofs
- *Not needed* for the existence of a proof

Essential:

- *Needed* for the existence of a proof:
  loop invariants, assignable clauses, ...

Requirement vs. Auxiliary Specification

Annotations serve different purposes

Requirement specification: Express the properties to be verified

Auxiliary annotations: Provide knowledge about the program
Support the verification process / finding a proof

Two kinds of auxiliary annotations

Needed for efficiency (lemmas):

- Allows shorter and/or easier to find proofs
- *Not needed* for the existence of a proof

Essential:

- *Needed* for the existence of a proof:
  loop invariants, assignable clauses, . . .

Requirement vs. Auxiliary Specification

Annotations serve different purposes

Requirement specification: Express the properties to be verified

Auxiliary annotations: Provide knowledge about the program
Support the verification process / finding a proof

Two kinds of auxiliary annotations

Needed for efficiency (lemmas):

- Allows shorter and/or easier to find proofs
- *Not needed* for the existence of a proof

Essential:

- *Needed* for the existence of a proof:
  loop invariants, assignable clauses, . . .

Conclusions IV: The Bottle-Neck Now

Bottle-neck now:

Specification (methodologies, formalisms) for

- useful low-level spec
- adequate abstract model
  (important for verification but also certification)

Conclusions IV: The Bottle-Neck Now

Karlsruhe Institute of Technology

Bottle-neck now:

Specification (methodologies, formalisms) for

- useful low-level spec
- adequate abstract model
  (important for verification but also certification)

Thank you
for your attention!