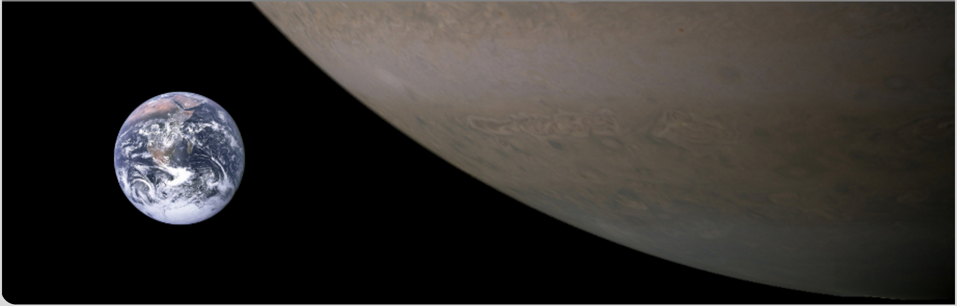
Scale Issues in Deductive Program Verification

Vladimir Klebanov | 9 March, 2011

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK



KeY Project

www.key-project.org

Deductive Verification of

- Java programs
- specified with the Java Modeling Language
- in Dynamic Logic

KeY Project

Karlsruhe Institute of Technology

www.key-project.org

Deductive Verification of

- Java programs
- specified with the
  Java Modeling Language
- in Dynamic Logic

KeY Tool

- Deductive rules for all Java features

KeY Project

www.key-project.org

Deductive Verification of

- Java programs
- specified with the
  Java Modeling Language
- in Dynamic Logic

KeY Tool

- Deductive rules for all Java features
- Symbolic execution

KeY Project

www.key-project.org

Deductive Verification of

- Java programs
- specified with the Java Modeling Language
- in Dynamic Logic

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card

KeY Project

www.key-project.org

Deductive Verification of
- Java programs
- specified with the Java Modeling Language
- in Dynamic Logic

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation/usability
  $>10,000$ loc / expert year

KeY Project

Karlsruhe Institute of Technology

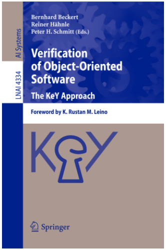
www.key-project.org

Deductive Verification of

- Java programs
- specified with the
  Java Modeling Language
- in Dynamic Logic

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- 100% Java Card
- High degree of automation/usability
  $>10,000$ loc / expert year



Verification Scalability So Far

What is the biggest system that can be verified (in unlimited time)?

http://en.wikipedia.org/wiki/File:All_Gizah_Pyramids.jpg

Verification Scalability So Far

Issues with this approach

- Hard to reproduce
- Hard to keep track of effort
- Usability swept under the rug

What is the biggest system that can be verified (in unlimited time)?

http://en.wikipedia.org/wiki/File:All_Gizah_Pyramids.jpg

Verification Scalability So Far

Issues with this approach

- Hard to reproduce
- Hard to keep track of effort
- Usability swept under the rug
- Needed: what can be specified and verified in 3h?

What is the biggest system that can be verified (in unlimited time)?

http://en.wikipedia.org/wiki/File:All_Gizah_Pyramids.jpg

1st Verified Software Competition

- informal event
- at VSTTE 2010 in Edinburgh
- organized by Peter Müller and Natarajan Shankar
- 5 problems (= pseudocode + informal spec + test cases)
- 4 hours of thinking time, 2 hours of hacking time
- no disciplines, no ranking

1st Verified Software Competition

- informal event
- at VSTTE 2010 in Edinburgh
- organized by Peter Müller and Natarajan Shankar
- 5 problems (= pseudocode + informal spec + test cases)
- 4 hours of thinking time, 2 hours of hacking time
- no disciplines, no ranking

The KeY Team: Vladimir Klebanov, Mattias Ulbrich, Benjamin Weiß

To Appear

The 1st Verified Software Competition: Experience Report

by

Peter Müller, Natarajan Shankar, Gary T. Leavens, Tom Ridge, Thomas Tuerk, Vladimir Klebanov, Mattias Ulbrich, Benjamin Weiß, K. Rustan M. Leino, Rod Chapman, Rosemary Monahan, Nadia Polikarpova, Derek Bronish, Rob Arthan, Eyad Alkassar, Ernie Cohen, Mark Hillebrand, Stephan Tobies, Bart Jacobs, Frank Piessens, Jan Smans

`www.vscomp.org`

Competing Tools

- **HOL4** (functional impl., spec in HOL)
- **ProofPower** (functional impl., spec in HOL)

- **Isabelle/VCG** (Hoare logic for C0)
- **Holfoot** (Separation logic for a C-like language, encoded in HOL)
- **KeY** (Dynamic logic for Java)
- **Dafny** (object-based language with built-in spec, like Java+JML)
- **SPARK/Ada** (contractualized subset of Ada)
- **Boogie** (intermediate language with assertions)
- **Resolve** (imperative component programs w/ modular specs)
- **VCC** (C with VCC assertions/invariants)
- **VeriFast** (Separation logic for Java and C)

Solution Overview

Tool	SUM&MAX	INVERT	LINKED-LIST	NQUEENS	QUEUE	SUM&MAX	INVERT	LINKED-LIST	NQUEENS	QUEUE	Team
Isabelle	■	□	■	□	□	■ a	□	■ a	□	□	A.Tsyban 1
HOL4	□	□	□	□	■	□	□	□	□	■ a	anonHolHacker 1
Holfoot	■	□	□	□	□	■	■	■	■	■	Holfoot 1
KeY	■	■	□	□	□	■ a	■ a	■	■	■	KeY 3
Dafny	■	□	■	□	■	■ a	■	■ a	■	■ a	Leino 1
SPARK	■	□	□	□	□	■ a	□	□	□	□	SparkULike 1
Boogie	■	□	◩	□	□	■ a	■	■	□	■	MonaPoli 2
Resolve	□	□	□	□	■ b	■	■	■	■	■ a	Resolve 1
ProofPower	■	□	■	□	□	■ a	□	■ a	□	□	RobArthan 1
VCC	■	□	■	□	□	■	■	■	■	■	VC Crushers 3
VeriFast	■	□	■	□	□	■	■	■ a	■	■	VeriFast 1

INVERT: Mathematically

The goal is to prove that for any $N > 0$, the injectivity of $\mathtt{B}$

$$\forall x, y.\ 0 \leqslant x < y < N \rightarrow \mathtt{B}[x] \neq \mathtt{B}[y] \qquad (1)$$

follows from the inverse relation between the arrays $\mathtt{A}$ and $\mathtt{B}$ (which per loop invariant holds after the loop)

$$\forall x.\ \big(0 \leqslant x < N \rightarrow \mathtt{B}[\mathtt{A}[x]] = x\big) \qquad (2)$$

and the surjectivity of $\mathtt{A}$ (which is a lemma that the problem description allowed to assume)

$$\forall x.\ \big((0 \leqslant x < N) \rightarrow \exists x'.\ (0 \leqslant x' < N) \wedge x = \mathtt{A}[x']\big)\ . \qquad (3)$$

INVERT: Mathematically

The g

Difficulties in this problem

- only interpreted arithmetical symbols in the quantifier guard
- required instantiations are Skolem constants

(1)

follow … B (which per lo…

$$\forall x.\ \big(0 \leqslant x < N \rightarrow \mathtt{B[A[}x\mathtt{]]} = x\big) \tag{2}$$

and the surjectivity of $\mathtt{A}$ (which is a lemma that the problem description allowed to assume)

$$\forall x.\ \big((0 \leqslant x < N) \rightarrow \exists x'.\ (0 \leqslant x' < N) \wedge x = \mathtt{A[}x'\mathtt{]}\big)\ . \tag{3}$$

INVERT: Mathematically

The g

(1)

follow (which

per loo

(2)

and th

descr

$$\forall x. \big((0 \leqslant x < N) \to \exists x'. (0 \leqslant x' < N) \land x = \mathrm{A}[x']\big) \quad . \tag{3}$$

Difficulties in this problem

- only interpreted arithmetical symbols in the quantifier guard
- required instantiations are Skolem constants

Range of solutions

- Manual instantiation
- Dummy function trigger
- Complex reformulations

Metric: Specification Verbosity

Tool	SUM&MAX			INVERT			LINKEDLIST		
HOL4	–	–	–	–	–	–	–	–	–
KeY	70	120	110	50	195	52+	90	151	233
Dafny	80	42	11	52	234	99	122	162	194
Boogie	84	12	12	58	125	458	82	315	41
Resolve	138	221	71	109	228	57	126	499	48
ProofPower	48	173	285	–	–	–	121	68	548
VCC	80	148	208	44	241	54	73	129	114
VeriFast	80	66	450	47	273	1834	59	94	359

Tokens of code / requirement annotations / proof guidance annotations

Metric: Specification Verbosity

Tool	SUM&MAX			INVERT			LINKEDLIST		
HOL4	–	–	–	–	–	–	–	–	–
KeY	70	120	110	50	195	52+	90	151	233
Dafny	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Boogie	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Resolve	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
ProofPower	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
VCC	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
VeriFast	80	66	450	47	273	1834	59	94	359

Grain of salt

- Parsimony is good.
- But so is: elegance, naturality, usefulness, ubiquity
- Different formalizations are hard to compare

Tokens of code / requirement annotations / proof guidance annotations

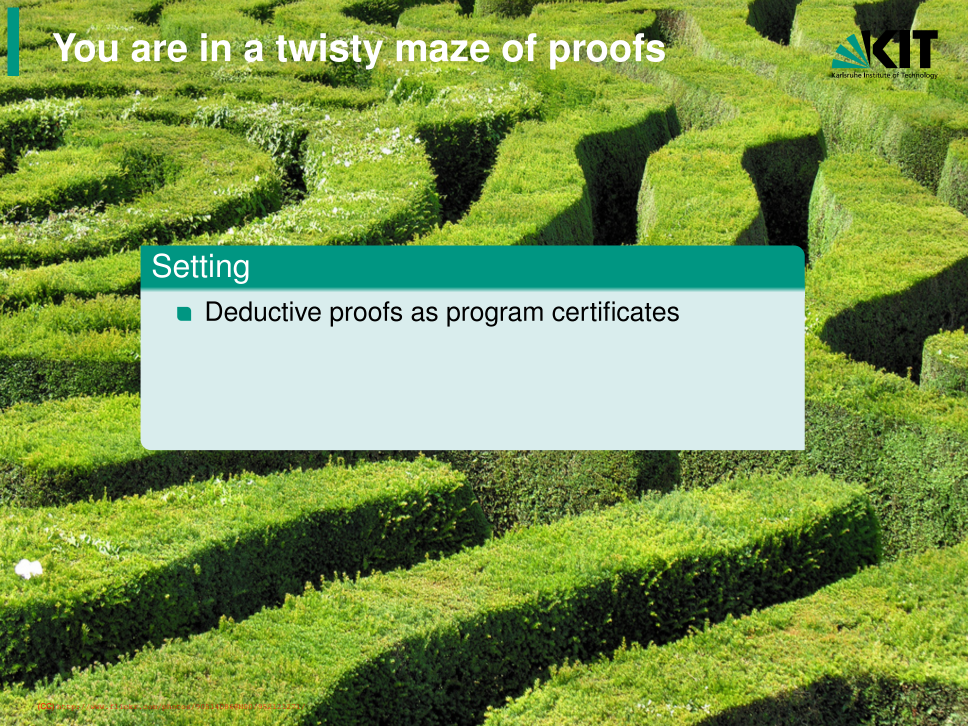
Conclusions

Karlsruhe Institute of Technology

- Issue: Control of SMT
- Issue: Abstract data types
- Degree of automation played hardly any role
- Performance played little role
- Benchmarking difficult—profile the user, not just the tool

You are in a twisty maze of proofs

You are in a twisty maze of proofs



Setting

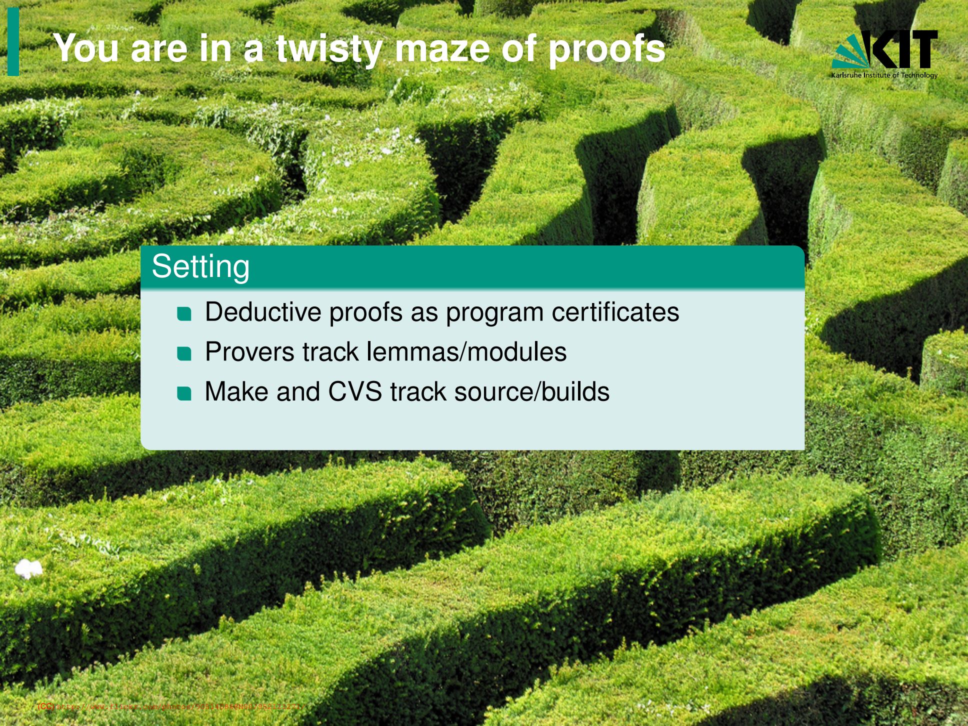
- Deductive proofs as program certificates

You are in a twisty maze of proofs

Setting

- Deductive proofs as program certificates
- Provers track lemmas/modules

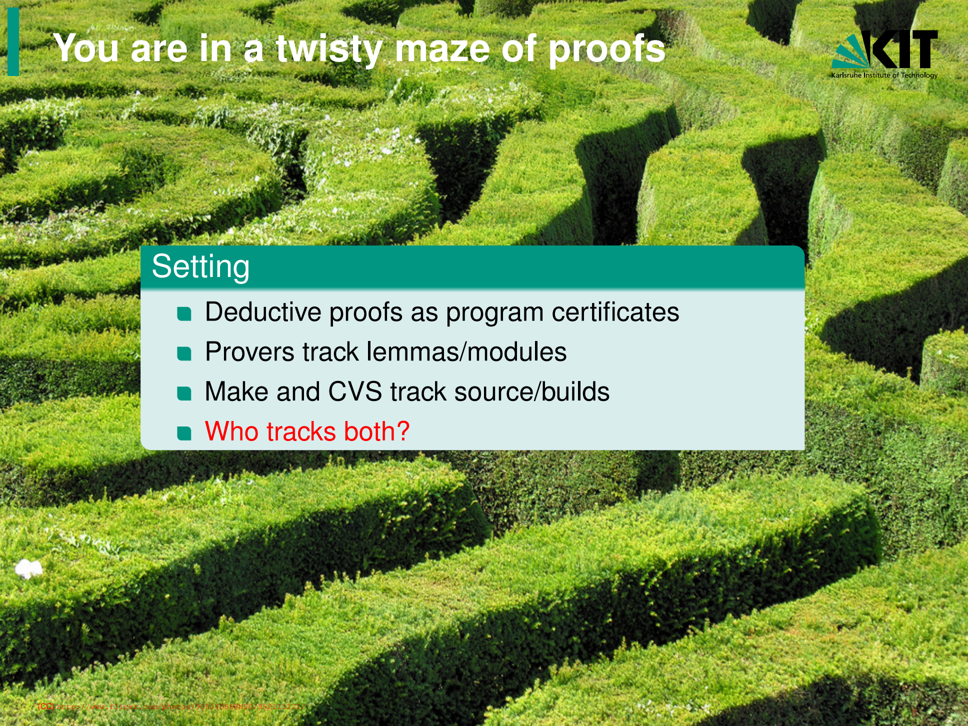
You are in a twisty maze of proofs



Setting

- Deductive proofs as program certificates
- Provers track lemmas/modules
- Make and CVS track source/builds

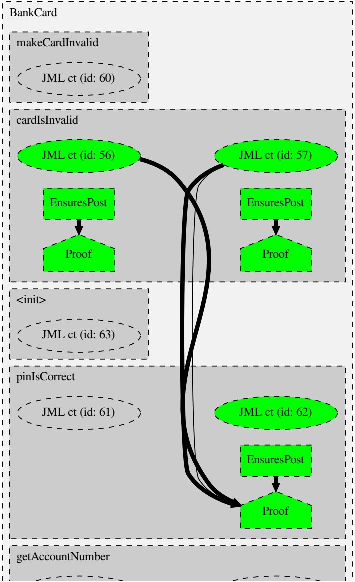
You are in a twisty maze of proofs



Setting

- Deductive proofs as program certificates
- Provers track lemmas/modules
- Make and CVS track source/builds
- Who tracks both?

Demo


BankCard
makeCardInvalid
JML ct (id: 60)
cardIsInvalid
JML ct (id: 56)
JML ct (id: 57)
EnsuresPost
Proof
EnsuresPost
Proof
<init>
JML ct (id: 63)
pinIsCorrect
JML ct (id: 61)
JML ct (id: 62)
EnsuresPost
Proof
getAccountNumber

You are in a twisty maze of products

You are in a twisty maze of products

Setting

A **product line** is a set of software systems (products) with well-defined commonalities and variabilities.

The Problem We Solve

Karlsruhe Institute of Technology

Given:

- a specified/verified product P_1
- a set of proofs for the product P_1
- an applicable delta set $\Delta(P_1, P_2)$

Wanted:

- a set of valid proofs for the product P_2
- …faster than (re-)verifying P_2 in isolation

A solution:

Proof slicing algorithm

with Daniel Bruns and Ina Schaefer [Formal Verification of OO Software 2010]

What's in a Delta?

- Add/remove **class**
- Change direct superclass (**reparent**)
- Add/remove **field**
- Add/remove **method**
- Add/remove **method contract**
- Add/remove **class invariant**

Slicing Algorithm (1): Adding Fields

For each *adds*(C::f):

1. find (statically) the set of method implementations M referring to C::f in the new product

Slicing Algorithm (1): Adding Fields

For each $adds(C{::}f)$:

1. find (statically) the set of method implementations M referring to $C{::}f$ in the new product
   - invalidate all pre-existing proofs about any $C'{::}m \in M$
   - invalidate all pre-existing proofs inlining any $C'{::}m \in M$

Slicing Algorithm (1): Adding Fields

For each $adds(C::f)$:

1. find (statically) the set of method implementations M referring to $C::f$ in the new product
   - invalidate all pre-existing proofs about any $C'::m \in M$
   - invalidate all pre-existing proofs inlining any $C'::m \in M$
2. invalidate all pre-existing proofs of specifications referring to $C::f$ in the new product

Slicing Algorithm (1): Adding Fields

For each $adds(C{::}f)$:

1. find (statically) the set of method implementations M referring to $C{::}f$ in the new product
   - invalidate all pre-existing proofs about any $C'{::}m \in M$
   - invalidate all pre-existing proofs inlining any $C'{::}m \in M$
2. invalidate all pre-existing proofs of specifications referring to $C{::}f$ in the new product

```
class C extends D {

   //@ invariant f == ((D)this).f;
}
```

Slicing Algorithm (1): Adding Fields

For each *adds*(C::f):

1. find (statically) the set of method implementations M referring to C::f in the new product
   - invalidate all pre-existing proofs about any $C'::m \in M$
   - invalidate all pre-existing proofs inlining any $C'::m \in M$
2. invalidate all pre-existing proofs of specifications referring to C::f in the new product

```
class C extends D {
    Object f;
    //@ invariant f == ((D)this).f;
}
```

Slicing Algorithm (1): Adding Fields

For each $adds(C{::}f)$:

1. find (statically) the set of method implementations M referring to $C{::}f$ in the new product
   - invalidate all pre-existing proofs about any $C'{::}m \in M$
   - invalidate all pre-existing proofs inlining any $C'{::}m \in M$
2. invalidate all pre-existing proofs of specifications referring to $C{::}f$ in the new product

```
class C extends D {
    Object f;
    //@ invariant f == ((D)this).f;
}
```

3. add non-nullness invariant for $C{::}f$

Slicing Algorithm (2): Adding Methods

For each *adds*(C::m):

1. invalidate all pre-existing proofs where m was inlined and C::m would have been a *relevant implementation* (mostly w.r.t. dynamic binding)

Slicing Algorithm (2): Adding Methods

For each *adds*(C::m):

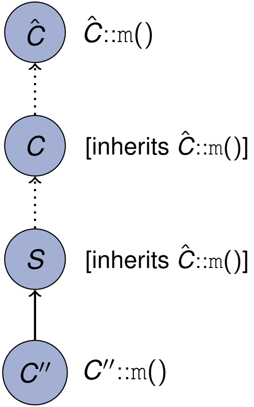
1. invalidate all pre-existing proofs where m was inlined and C::m would have been a *relevant implementation* (mostly w.r.t. dynamic binding)
2. proofs using the contracts for m remain valid

Slicing Algorithm (2): Adding Methods

For each *adds*(C::m):

1. invalidate all pre-existing proofs where m was inlined and C::m would have been a *relevant implementation* (mostly w.r.t. dynamic binding)
2. proofs using the contracts for m remain valid
3. prove that C::m satisfies all specifications of C (either stated directly or inherited), as well as all other invariants

Relevant Method Implementations

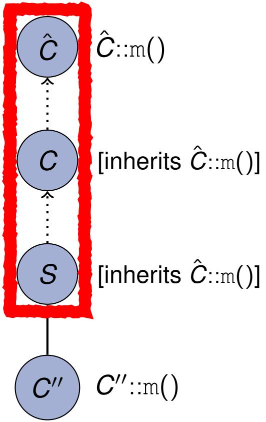



```
class A {
    //@ ensures \result > 0;
    int foo() {
        return 23;
}   }

class B extends A {


}
```

Relevant Method Implementations

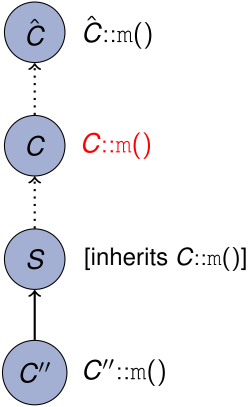



```
class A {
    //@ ensures \result > 0;
    int foo() {
        return 23;
}   }

class B extends A {


}
```

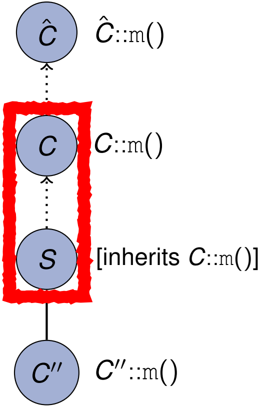
Relevant Method Implementations




```
class A {
    //@ ensures \result > 0;
    int foo() {
        return 23;
}   }

class B extends A {
    int foo() {
        return 42;
}   }
```

Relevant Method Implementations




```
class A {
    //@ ensures \result > 0;
    int foo() {
        return 23;
}   }

class B extends A {
    int foo() {
        return 42;
}   }
```

Slicing Algorithm (3): Class Reparenting

For each *reparents*(C, C'):

1. invalidate all pre-existing proofs inlining any $C''::m$ with $C'' \sqsubseteq C$

Slicing Algorithm (3): Class Reparenting

For each *reparents*(C, C'):

1. invalidate all pre-existing proofs inlining any $C''::m$ with $C'' \sqsubseteq C$
2. contracts for methods in reparented classes remain valid unless the contract no longer exists (inherited contract)

Slicing Algorithm (3): Class Reparenting

For each *reparents*(C, C'):

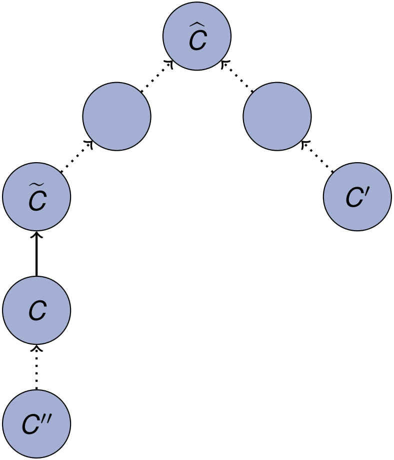
1. invalidate all pre-existing proofs inlining any $C''::m$ with $C'' \sqsubseteq C$
2. contracts for methods in reparented classes remain valid unless the contract no longer exists (inherited contract)
3. invalidate proofs for specifications inherited from any class K with $\widetilde{C} \sqsubseteq K \sqsubset \widehat{C}$ where $\widehat{C}$ is the least common supertype of C' and the old direct supertype $\widetilde{C}$ of C

Slicing Algorithm (3): Class Reparenting

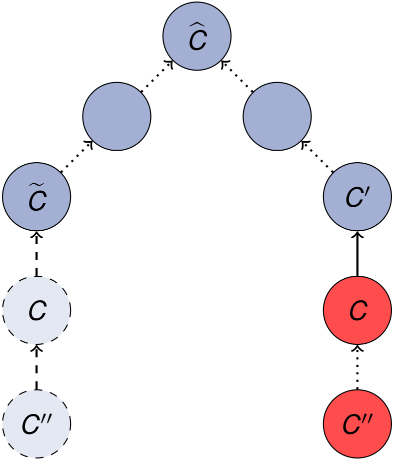
For each *reparents*(C, C'):

1. invalidate all pre-existing proofs inlining any $C''::m$ with $C'' \sqsubseteq C$
2. contracts for methods in reparented classes remain valid unless the contract no longer exists (inherited contract)
3. invalidate proofs for specifications inherited from any class K with $\widetilde{C} \sqsubseteq K \sqsubset \widehat{C}$ where $\widehat{C}$ is the least common supertype of C' and the old direct supertype $\widetilde{C}$ of C
4. prove that all classes $C'' \sqsubseteq C$ satisfy the specifications inherited from new superclasses K with $C' \sqsubseteq K \sqsubset \widehat{C}$

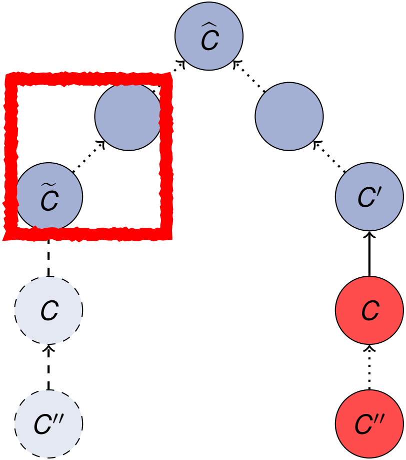
Slicing Algorithm (3): Class Reparenting



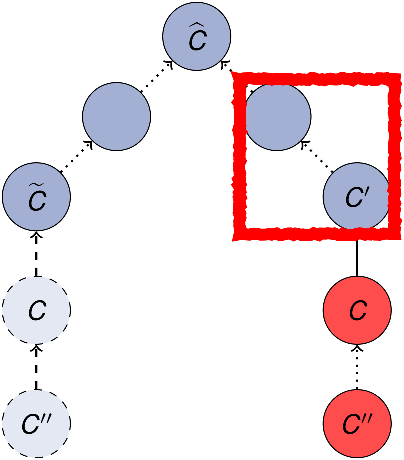
Slicing Algorithm (3): Class Reparenting



Slicing Algorithm (3): Class Reparenting



Slicing Algorithm (3): Class Reparenting



2nd Step: Proof Reuse

Karlsruhe Institute of Technology

Idea

- Some proofs have been killed in slicing
- Still, new proofs for product P_2 often similar to those in P_1
- Solution: similarity-guided proof reuse [SEFM 2004]

Proof reuse in KeY

- Originally implemented to support incremental software development
- ... in interactive verification
- Sound by design

Not Tied to One Verification System

Karlsruhe Institute of Technology

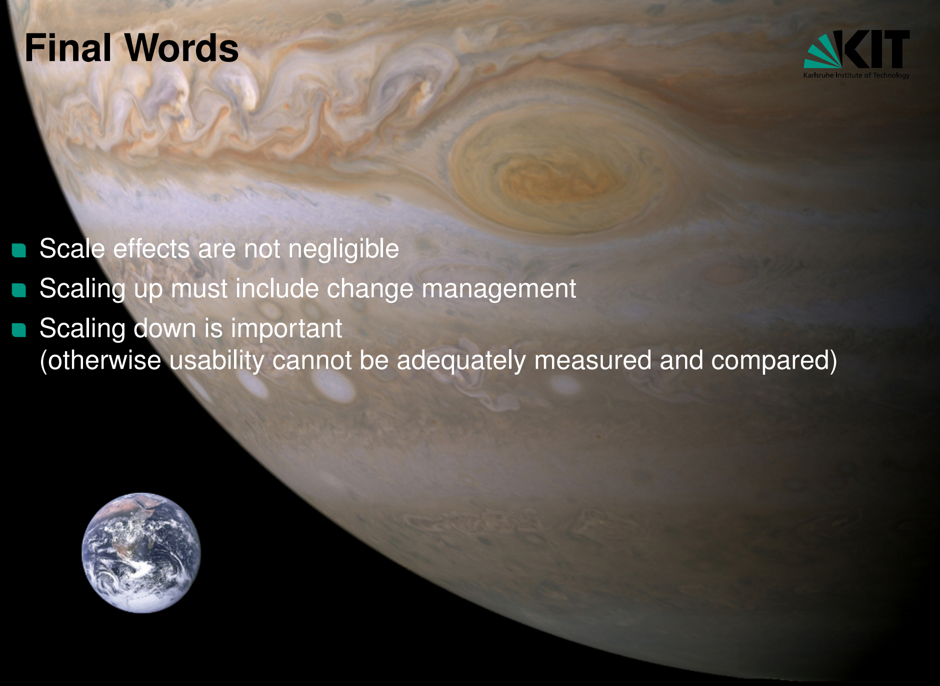
- We do assume syntax-correct, typesafe products
- Method calls by contract or inlining
- Parametric invariant checking
- Conservative proof invalidation (currently based on structural change information only)

Warning

JML-style specifications and code are not separated.
Changes to code may not mean what you think they mean.

Final Words

- Scale effects are not negligible
- Scaling up must include change management
- Scaling down is important
  (otherwise usability cannot be adequately measured and compared)



http://en.wikipedia.org/wiki/File:Jupiter-Earth-Spot_comparison.jpg