Scale Issues in Deductive Program Verification

Vladimir Klebanov | 9 March, 2011
Deductive Verification of
- Java programs
- specified with the Java Modeling Language
- in Dynamic Logic
Deductive Verification of
- Java programs
- specified with the
  Java Modeling Language
- in Dynamic Logic

KeY Tool
- Deductive rules for all Java features
KeY Project

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

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
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

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

Issues with this approach

- Hard to reproduce
- Hard to keep track of effort
- Usability swept under the rug
Verification Scalability So Far

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

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?
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

<table>
<thead>
<tr>
<th>Tool</th>
<th>SUM&amp;MAX</th>
<th>INVERT</th>
<th>LINKED-LIST</th>
<th>N-QUEENS</th>
<th>QUEUE</th>
<th>SUM&amp;MAX</th>
<th>INVERT</th>
<th>LINKED-LIST</th>
<th>N-QUEENS</th>
<th>QUEUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isabelle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOL4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>Holfoot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dafny</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPARK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boogie</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProofPower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VeriFast</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Team**

- A.Tsyban
- anonHOLHacker
- Holfoot
- KeY
- Leino
- SparkULike
- MonaPoli
- Resolve
- RobArthan
- VC Crushers
- VeriFast

*Vladimir Klebanov – Scale Issues in Deductive Program Verification*

*9 March, 2011*
The goal is to prove that for any $N > 0$, the injectivity of $B$

$$\forall x, y. \ 0 \leq x < y < N \rightarrow B[x] \neq B[y]$$

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

$$\forall x. \ (0 \leq x < N \rightarrow B[A[x]] = x)$$

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

$$\forall x. \ ((0 \leq x < N) \rightarrow \exists x'. \ (0 \leq x' < N) \land x = A[x'])$$
The goal is to prove that for any \( N > 0 \), the injectivity of \( B \) follows from the inverse relation between the arrays \( A \) and \( B \) (which per loop invariant holds after the loop):

\[
\forall x. \ (0 \leq x < N \rightarrow B[A[x]] = x) \quad (2)
\]

and the surjectivity of \( A \) (which is a lemma that the problem description allowed to assume):

\[
\forall x. \ ((0 \leq x < N) \rightarrow \exists x'. \ (0 \leq x' < N) \land x = A[x']) \quad (3)
\]
**INVERT: Mathematically**

The goal is to prove that for any $N > 0$, the injectivity of $B$ follows from the inverse relation between the arrays $A$ and $B$ (which per loop invariant holds after the loop):

$\forall x. \left( 0 \leq x < N \rightarrow B[A[x]] = x \right)$ \hspace{1cm} (2)

**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

$\forall x. \left( (0 \leq x < N) \rightarrow \exists x'. (0 \leq x' < N) \land x = A[x'] \right)$ \hspace{1cm} (3)
## Metric: Specification Verbosity

<table>
<thead>
<tr>
<th>Tool</th>
<th>Sum&amp;Max</th>
<th>Invert</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOL4</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>KeY</td>
<td>70 120 110</td>
<td>50 195 52+</td>
<td>90 151 233</td>
</tr>
<tr>
<td>Dafny</td>
<td>80 42 11</td>
<td>52 234 99</td>
<td>122 162 194</td>
</tr>
<tr>
<td>Boogie</td>
<td>84 12 12</td>
<td>58 125 458</td>
<td>82 315 41</td>
</tr>
<tr>
<td>Resolve</td>
<td>138 221 71</td>
<td>109 228 57</td>
<td>126 499 48</td>
</tr>
<tr>
<td>ProofPower</td>
<td>48 173 285</td>
<td>– – –</td>
<td>121 68 548</td>
</tr>
<tr>
<td>VCC</td>
<td>80 148 208</td>
<td>44 241 54</td>
<td>73 129 114</td>
</tr>
<tr>
<td>VeriFast</td>
<td>80 66 450</td>
<td>47 273 1834</td>
<td>59 94 359</td>
</tr>
</tbody>
</table>

Tokens of code / requirement annotations / proof guidance annotations
## Metric: Specification Verbosity

<table>
<thead>
<tr>
<th>Tool</th>
<th>SUM&amp;MAX</th>
<th>INVERT</th>
<th>LINKEDLIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOL4</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>KeY</td>
<td>70 120 110</td>
<td>50 195 52+</td>
<td>90 151 233</td>
</tr>
<tr>
<td>Dafny</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boogie</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolve</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProofPower</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCC</td>
<td>80 448 205</td>
<td>47 273 1834</td>
<td>59 94 359</td>
</tr>
<tr>
<td>VeriFast</td>
<td>80 66 450</td>
<td>47 273 1834</td>
<td>59 94 359</td>
</tr>
</tbody>
</table>

**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

- 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?
You are in a twisty maze of products
You are in a twisty maze of products

<table>
<thead>
<tr>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>A <strong>product line</strong> is a set of software systems (products) with well-defined commonalities and variabilities.</td>
</tr>
</tbody>
</table>
The Problem We Solve

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 \( \textit{adds}(C::f) \):

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 \( C':m \in M \)
For each $\textit{adds}(C::f)$:

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

```java
class C extends D {
    //@ invariant f == ((D)this).f;
}
```
Slicing Algorithm (1): Adding Fields

For each \textit{adds}(C::f):
\begin{enumerate}
\item find (statically) the set of method implementations \(M\) referring to \(C::f\) in the new product
  \begin{itemize}
  \item invalidate all pre-existing proofs about any \(C'::m \in M\)
  \item invalidate all pre-existing proofs inlining any \(C'::m \in M\)
  \end{itemize}
\item invalidate all pre-existing proofs of specifications referring to \(C::f\) in the new product
\end{enumerate}

```java
class C extends D {
    Object f;
    //@ invariant f == ((D)this).f;
}
```
Slicing Algorithm (1): Adding Fields

For each \textit{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

3. add non-nullness invariant for \( C::f \)

```java
class C extends D {
    Object f;
    //@ invariant f == ((D)this).f;
}
```
Slicing Algorithm (2): Adding Methods

For each $\textit{adds}(C::m)$:

1. Invalidate all pre-existing proofs where $m$ was inlined and $C::m$ would have been a \textit{relevant implementation} (mostly w.r.t. dynamic binding)
Slicing Algorithm (2): Adding Methods

For each \( \textit{adds}(C::m) \):

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

2. proofs using the contracts for \( m \) remain valid
Slicing Algorithm (2): Adding Methods

For each \textit{adds}(C::m):

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

2. proofs using the contracts for \textit{m} remain valid

3. prove that \textit{C::m} satisfies all specifications of \textit{C} (either stated directly or inherited), as well as all other invariants
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 {
   ...
}
class A {
    //@ ensures \result > 0;
    int foo() {
        return 23;
    }
}

class B extends A {
    int foo() {
        return 42;
    }
}
class A {
    //@ ensures \result > 0;
    int foo() {
        return 23;
    }
}

class B extends A {
    int foo() {
        return 42;
    }
}
Slicing Algorithm (3): Class Reparenting

For each \textit{reparents}(C, C'):

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

Vladimir Klebanov – Scale Issues in Deductive Program Verification

9 March, 2011
Slicing Algorithm (3): Class Reparenting

For each \textit{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)
For each \textit{reparents}(C, C'):

1. Invalidate all pre-existing proofs inlining any $C'' : \vdash m$ 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 $\tilde{C} \sqsubseteq K \sqsubseteq \hat{C}$ where $\hat{C}$ is the least common supertype of $C'$ and the old direct supertype $\tilde{C}$ of $C$
Slicing Algorithm (3): Class Reparenting

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

1. invalidate all pre-existing proofs inlining any $C''::m$ with $C'' \subseteq C$
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 $\tilde{C} \subseteq K \subseteq \hat{C}$ where $\hat{C}$ is the least common supertype of $C'$ and the old direct supertype $\tilde{C}$ of $C$
4. prove that all classes $C'' \subseteq C$ satisfy the specifications inherited from new superclasses $K$ with $C' \subseteq K \subseteq \hat{C}$
Slicing Algorithm (3): Class Reparenting
Slicing Algorithm (3): Class Reparenting
Slicing Algorithm (3): Class Reparenting

Diagram showing the relationships between classes and their reparenting in a sliced program.
Slicing Algorithm (3): Class Reparenting
2\textsuperscript{nd} Step: Proof Reuse

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

- 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)