Computer-Assisted Proving for the Analysis of Programs and Specifications

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at
1. The RISC Institute

2. Formal Methods at RISC

3. A Short Review of Verification Techniques

4. Computer-Assisted Proving for Program Analysis

5. Conclusions
The RISC Institute

- **Research Institute for Symbolic Computation (RISC)**
  *Institut für Symbolisches Rechnen*
  Current chair is Prof. Franz Winkler.
- 16 faculty members.
  3 full professors, 4 associate professors, 3 adjunct professors.
- 22 PhD students.
  About 80 PhDs graduated from RISC.
- An institute of the **Johannes Kepler University (JKU)**
  *Johannes Kepler Universität Linz, Austria*
- 14,000 students and 2,000 employees.
- 29 bachelor, master, and doctoral studies.
- Since 1989 located in the **Castle of Hagenberg**
  Schloss Hagenberg, Hagenberg im Mühlkreis.
- About 20 km from Linz.

http://www.risc.uni-linz.ac.at
How to Find Hagenberg

Just on the way from Prague to Linz!
The Castle of Hagenberg
RISC in Hagenberg

Various institutions arose by initiatives of Prof. Buchberger (RISC).

- Softwarepark Hagenberg
  - Industrial park with 40 companies (1000 employees).
- University of Applied Sciences (FH) Campus Hagenberg
  - Informatics, communications and media (1200 students).
- Software Competence Center Hagenberg (SCCH)
  - Joint academic/industrial R&D (70 developers).
- High School (BORG) for Communication in Hagenberg
  - Language, communication, technology (100 students).
- International School for Informatics Hagenberg (ISI)
  - International master program (30 students).

Today, in Hagenberg about 2500 people study/work in IT-related topics.

Wolfgang Schreiner
http://www.risc.uni-linz.ac.at
Hagenberg in 2008
RISC Research

- **Computer Mathematics** with three main themes.
  - **Computer Algebra**
    - Algebraic geometry and algorithmic combinatorics.
    - Modeling and solving problems from mathematics, physics, biology.
  - **Computational Logic**
    - Automated theorem proving, mathematical theory exploration.
    - Formal modeling and verification of computer programs and systems.
  - **Mathematical Software**
    - Components for computer algebra systems and theorem provers.
    - Parallel, distributed, grid computing software.

- **Sample Activities**
RISC Summer 2009

RISC SUMMER 2009
Research Institute for Symbolic Computation
Johannes Kepler University Linz

The Fourth RISC/SCIence Training School in Symbolic Computation, June 29–July 10

CADGME 2009
Computer Algebra and Dynamic Geometry Systems in Mathematics Education, July 11–13

AACA
Algebraic Analysis and Computer Algebra – New Perspectives for Applications,
July 13–17

GeoGebra 2009
1st International GeoGebra Conference, July 14–15

WWW 2009
5th International Workshop on Automated Specification and Verification of
Web Systems, July 17

FRSAC 2009
21st International Conference on Formal Power Series and
Algebraic Combinatorics, July 20–24

+Combinat 2009
International Sage Workshop on Free and Practical Software for
Algebraic Combinatorics, July 25–29

Castle of Hagenberg, Austria
June 29 – July 29, 2009

http://www.risc.uni-linz.ac.at/conferences/summer2009

RISC Summer Series:
Science and More
RISC Education

- JKU Master Program “Computer Mathematics”
  - One of three JKU master programs on mathematics.
- RISC PhD Program on Symbolic Computation
  - Every year 3–5 new international students accepted.
  - Master in mathematics or computer science.
  - 3–4 years of study.
  - Training semester, course work, participation in research project, publications, PhD thesis.
- Funding from Upper Austria (initial year) and then from participation in funded research projects.

- Doctoral Program “Computational Mathematics”
  - Several mathematics institutes of the JKU.
  - Funded from Austrian Science Fund and Upper Austria (2008–)
RISC Industry

- **RISC Software Company**
  - [http://www.risc-software.at](http://www.risc-software.at)
  - Since 2004/2008 owned by JKU and Upper Austrian Research (UAR).
  - 40 employees, 2.3 Mio Euro turnover (2007).

- **Logistics Informatics**
  - Software for distribution logistics, warehouse control, production process planning.

- **Industrial Computation**
  - Scientific computing, optimization, computational geometry, visualization, parallel and distributed/grid computing.

- **Medical Informatics**
  - Medical simulation and diagnosis/decision support, medical documentation and knowledge management.
1. The RISC Institute

2. Formal Methods at RISC

3. A Short Review of Verification Techniques

4. Computer-Assisted Proving for Program Analysis

5. Conclusions
A Selection of PhD Theses

1974 Jenewein, Franz: Eine Metasprache zur Definition von Programmiersprachen und ihre Implementierung
1976 Sutter, Peter: Implementierung einer Metasprache zur Definition von auf symbolischen Daten operierenden Interpretern und ihr Einsatz in der Programmverifikation
1989 Funk, Gerhard: A Verification System for Total Correctness of the Parallel L-Language
2000 Dupre, Daniela: Automated Theorem Proving by Integrating Proving, Solving and Computing
2007 Kovacs, Laura: Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema
2008 Popov, Nikolaj: Functional Program Verification in Theorema

Much more related work in the context of computational logic and automated theorem proving.
Computer Mathematics

A core focus: the relationship between computers and mathematics/logic.

- **Mathematical/logic formalisms for programming.**
  
  *The computational capabilities of mathematics/logic.*

- **Declarative (functional, logic, constraint) programming languages and executable specifications of abstract datatypes.**
  
  Automated termination proofs for recursive functional programs.

- **Algorithm synthesis from formal problem specifications.**
  
  Synthesis of Buchberger’s Gröbner bases algorithm.

- **Programs for solving mathematical/logical problems.**
  
  *The mathematical/logical capabilities of computers.*

  - Symbolic solutions of polynomial equation systems over \( \mathbb{C} \) or \( \mathbb{R} \).
    
    Buchberger’s Gröbner bases algorithm and others.

  - Symbolic solutions of recurrences, summations, integrals.
    
    Generation of program invariants by algebraic techniques.

**Two dual (mutually recursive) aspects of “computer mathematics”.**
Termination of Functional Programs

- **Problem:** (dis)prove termination of functional program $\mathcal{F}(x)$.
  - $\mathcal{F}$ is in general a set of mutually recursive function definitions.
  - Every function is annotated by an input/output specification.
- **Approach:** combination of heuristics and algebraic techniques.
  - Construct "termination pattern" (predicate) $\mathcal{P}_\mathcal{F}(x)$.
  - Lookup $\mathcal{P}_\mathcal{F}[x]$ in library of already proved patterns.
    - Many functions yield the same termination pattern.
  - Correctness of each pattern proved by computer algebra algorithms
    - for transforming recurrence $x_n[x_{n-1}]$ (describing the value of parameter $x$ after $n$ recursive invocations) to explicit solution $x_n[n, x_0]$,
    - for finding, for initial argument $x_0$, the smallest $n$ such that the pattern’s exit condition $\mathcal{Q}[x_n[n, x_0]]$ is satisfied.
- **Example:** algorithm for integer approximation of cubic root.

  $$a[x, s, r] = \text{if } x \leq s \text{ then } \langle x, s, r \rangle \text{ else } a[x - s, s + 6r + 3, r + 1]$$


Algorithm Synthesis by “Lazy Thinking”

- **Problem**: construct algorithm satisfying a given specification.
  - Given a formal theory of the symbols used in the formalization.
- **Approach**: combine heuristics and automated proving techniques.
  - Choose algorithmic scheme from library of pre-defined schemes.
    - An algorithm matter with unspecified subalgorithms.
  - Attempt automated proof of the correctness of the algorithm.
    - Generates verification conditions and attempts to prove these.
  - Automatically analyze failed proof situations, generate conditions on the subalgorithms that are sufficient to overcome the failure.
    - Repeat process until proof is completed.
- Generated conditions represent specifications of subalgorithms.
  - Retrieve these from library of existing algorithms or synthesize them.
- **Implemented in Buchberger’s Theorema system.**

- **Example**: synthesis of Buchberger’s Gröbner bases algorithm.


Wolfgang Schreiner
Invariant Generation by Computer Algebra

- **Problem:** construct an invariant for a given loop.
  - Condition on program state preserved by every loop iteration.
  - Condition shall be “as strong as possible”.
    - May over-approximate the set of reachable states.
- **Approach:** computer algebra and algorithmic combinatorics.
  - Generate system of recurrence equations on loop variables.
  - Solve recurrence equations yielding polynomial equations.
  - Eliminate iteration counters and temporary variables from equations.
- **Example:** Wensley’s algorithm for real division.

  ```plaintext
  while d ≥ T do
    if P < a + b then
      b := b/2; d := d/2
      a := a + b; y := y + d/2; b := b/2; d := d/2
  ```


Problem: turn mathematical programs into web services.
- Embed them into appropriate execution framework.
- Allow them to be automatically discovered and used by clients.

Approach: formal specification and automated proving.
- Mathematical Service Description Language (MSDL).
- Declarative specification of mathematical functionality.
  - Broker matches formally described problem solution requests to library of service specifications with the use of an automatic prover.
- Process-algebraic description of interaction protocol.
  - Generic client engine reads and interprets descriptions.

Example: QEPCAD software.
Quantifier Elimination by Partial Cylindrical Algebraic Decomposition.


Performance Analysis by Model Checking

Joint work with universities of Debrecen and Eger.

- **Goal:** compare approaches to performance modeling and analysis.
  - Queuing theory (classical): queueing networks model quantitative system aspects.
  - Probabilistic model checking (newer): state models with probabilistic transitions cover both qualitative and quantitative aspects.
- **Results:** probabilistic model checking is an attractive alternative.
  - State models allow validation of the adequacy of the models.
  - Implementation details (buffer sizes) have to be made explicit.
  - Previously published web server models were partially inadequate, based on misunderstandings, and even contained plain errors.


Formally Specified Computer Algebra

- **Problem:** increase trust in computer algebra software.
  - Written in dynamically typed scripting languages.
    - Computer algebra systems like Mathematica, Maple, ...
  - Mostly functional interfaces, mostly imperative implementations.
  - Behavior poorly specified, results often require human interpretation.
- **Approach:** apply formal methods technology.
  - Formal contract specifications.
    - Impose static type system based on abstract datatypes.
    - Model-based reasoning for programs operating on concrete type representations.
  - Static type checking.
  - Runtime assertion checking.
  - Static verification (especially of method preconditions).
- **Start:** middle of 2009.
  A project in the frame of the DK “Computational Mathematics”.

1. The RISC Institute

2. Formal Methods at RISC

3. A Short Review of Verification Techniques

4. Computer-Assisted Proving for Program Analysis

5. Conclusions
Verification by Model Checking

Today “verification” is almost a synonym for “model checking”.

- **Model Checking (MC):** investigate finite graph that models system.
  - Nodes represent states, edges represent state transitions.
- **Explicit State MC:** each node represents a single state.
  - Feasible by state space reduction and state compression.
  - Example: SPIN, Verisoft, etc.
- **Symbolic MC:** each node may represent a finite set of states.
  - Representation by efficient encodings for sets of bit vectors.
  - Example: SMV and successors, etc.
- **Abstraction:** each node may represent an infinite set of states.
  - Allows to model check infinite state systems.
    - Represented by the truth values of a chosen set of predicates.
    - Example: BLAST, Magic, SLAM, etc.

Software model checking today typically employs abstraction techniques.
Specifications for Model Checking

Which property of the model is checked?

- **Reachability of certain (error) states**
  - Explicitly marked in the program.
    
  ```
  if (x == 0) { ERROR: goto ERROR; }
  ```
  - Derived from an assertion in the program.
    
  ```
  assert(x != 0);
  ```
  - Taken from a monitoring automaton external to the program.
    
  ```
  state { int c = 0; }
  put.entry { if ($1 == 0) { if (c >= N) abort "full"; else c++; } }
  get.exit { if ($return == 0) c--; }
  ```

- **Conformance of executions to protocols**

  Based on events/atomic formulas defined by the programmer.

  - Specified by a monitoring automata.
    
    ```
    S1: if :: (! ((e_put)) && (x_put)) -> goto S2 :: (1) -> goto S1; fi;
    S2: if :: (! ((x_put))) -> goto S2 fi;
    ```

  - Specified by a regular expression.
    
    ```
    ( (enter_put; exit_put) | (enter_get; exit_get) )[*]
    ```

  - Specified by a propositional temporal logic formula.
    
    ```
    []( enter_put -> (!enter_get) U exit_put )
    ```

Basic properties have to be specified as executable checks.
Results of Model Checking

Given a system and a property, what does model checking give us?

- **A fully automatic answer** (“valid”/”invalid”)
  - No/little user assistance required.
    - Perhaps appropriate selection of optimization options.
    - Perhaps help in the selection of abstraction predicates.
  - Checking overhead directly depends on size of investigated graph.
    - Counter-example guided abstraction refinement (CEGAR): number of refinement iterations required.

- **A counterexample trace** (if answer is “invalid”)
  - Sequence of instructions that leads to invalid state respectively proves non-conformance to protocol.

```c
int foo(int x, int y) { if (x > y) { x = y - x; assert(x > 0); } }
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4: Pred(x@foo&gt;y@foo) :: 5</td>
</tr>
<tr>
<td>5</td>
<td>5: Block(x@foo = y@foo - x@foo;) :: 6</td>
</tr>
<tr>
<td>6</td>
<td>6: Pred(Not (x@foo&gt;0)) :: 6</td>
</tr>
<tr>
<td>6</td>
<td>6: FunctionCall(__assert(&quot;foo.c&quot;, 6, &quot;x &gt; 0&quot;)) :: -1</td>
</tr>
</tbody>
</table>
Counterexample Trace

Simulation Output

```
48: proc 3 (server) line 25 "pan_in" (state 1) [values: 2?MESSAGE]  
48: proc 3 (server) line 25 "pan_in" (state 4) [request[1]?MESSAGE]  
50: proc 3 (server) line 26 "pan_in" (state 5) [sender = 2]  
52: proc 3 (server) line 58 "pan_in" (state 17) [given == 0]  <
54: proc 3 (server) line 41 "pan_in" (state 18) [given = sender]  
54: proc 3 (server) line 41 "pan_in" (state 19) [values: 4?MESSAGE]  
54: proc 3 (server) line 41 "pan_in" (state 19) [answer[1]]  !MESSAGE  
56: proc 3 (server) line 21 "pan_in" (state 1) [11]  
58: proc 2 (client) line 10 "pan_in" (state -) [values: 4!MESSAGE]  
60: proc 2 (client) line 10 "pan_in" (state 10) [answer[1]]  !MESSAGE  
62: proc 2 (client) line 12 "pan_in" (state -) [values: 2!MESSAGE]  
64: proc 2 (client) line 12 "pan_in" (state 10) [request[1]]  !MESSAGE  
66: proc 3 (server) line 25 "pan_in" (state 1) [(11)]  
68: proc 3 (server) line 25 "pan_in" (state 4) [request[1]]  !MESSAGE  
68: proc 3 (server) line 25 "pan_in" (state 5) [sender = 2]  
70: proc 3 (server) line 58 "pan_in" (state 17) [given == 0]  <
72: proc 3 (server) line 58 "pan_in" (state 17) [(11)]  
74: proc 3 (server) line 21 "pan_in" (state 1) [(11)]  
76: proc 2 (client) line 9 "pan_in" (state -) [values: 2!MESSAGE]  
78: proc 2 (client) line 9 "pan_in" (state 2) [request[1]]  !MESSAGE  
```

Sequence Chart

"Spin: trail ends after 76 steps"

"#processes: 4"

"proc 3 (server) line 22 "pan_in" (state 6)"

"proc 2 (client) line 10 "pan_in" (state 2)"

"proc 1 (client) line 10 "pan_in" (state 3)"

"proc 0 (init) line 53 "pan_in" (state 4)"

"4 processes created"

"Exit Status 0"
Applicability of Model Checking

So where is model checking applicable best?

- **Systems are finite or can be appropriately abstracted to such.**
  - Automation by CEGAR often (but not always) works.
- **Goal is finding automatically derivable error states.**
  - Automatic construction of corresponding assertions from program.
- Null-pointer dereferences, index-out-of-bound violations, division by zero, deadlocks, ...
- **Goal is checking for structurally simple properties.**
  - Can can be easily expressed in the specification language.
- Basic predicates simply expressible as constructive checks.
  - Relationship to previous state only by “history variables”.
- No infinite quantification is needed.
  - Finite quantification can be checked by loops (cumbersome).
- **Error traces give real insight** (for solving the core problem).
  - Typical: sequence of messages exchanged by concurrent processes.
  - Far less: sequences of local instructions.

Modeling/specifying may be awkward, error traces may be unhelpful.
// check m is minimum of array a of length n
void assert_minimum(int *a, int n, int m) {
    int i;
    int found = 0;
    for (i=0; i<n; i++) {
        assert(m <= a[i]);
        if (m == a[i]) found = 1;
    }
    assert(found == 1);
}

int minimum(int* a, int n) {
    int i;
    int m = a[0];
    for (i=1; i<n; i++) {
        if (a[i] < m) m = a[0]; // wrong
    }
    assert_minimum(a, n, m); // check function’s postcondition
    return m;
}

Correctness property expressed in an operational way.
Model-Checking

BLAST (http://mtc.epfl.ch/software-tools/blast)

32 :: 32: FunctionCall(tmp@main = minimum(a@minimum = a@main,n@minimum = n@main,)) :: -1
18 :: 18: Block(mem_temp5@minimum = a@minimum offset 0;m@minimum = *(mem_temp5@minimum );i@minimum = 1;) :: 19
19 :: 19: Pred(i@minimum < n@minimum) :: -1
21 :: 21: Block(mem_temp6@minimum = a@minimum offset i@minimum;) :: 21
21 :: 21: Pred(* (mem_temp6@minimum ) >= m@minimum) :: -1
19 :: 19: Block(i@minimum = i@minimum + 1;) :: 19
19 :: 19: Pred(i@minimum < n@minimum) :: -1
21 :: 21: Block(mem_temp6@minimum = a@minimum offset i@minimum;) :: 21
21 :: 21: Pred(* (mem_temp6@minimum ) >= m@minimum) :: -1
19 :: 19: Block(i@minimum = i@minimum + 1;) :: 19
19 :: 19: Pred(i@minimum < n@minimum) :: -1
...
23 :: 23: FunctionCall(check(a@check = a@minimum,n@check = n@minimum,m@check = m@minimum,)) :: -1
6 :: 6: Block(found@check = 0;i@check = 0;) :: 7 u
7 :: 7: Pred(i@check < n@check) :: -1
9 :: 9: Block(mem_temp6@check = a@check offset i@check;) :: 9
9 :: 9: Pred(m@check > * (mem_temp6@check )) :: -1
9 :: 9: FunctionCall(_-_assert_fail(_-_assertion@_-_assert_fail = "m <= a[i]",_-_file@_-_assert_fail = "m",...
77 :: 77: FunctionCall(_-_blast_assert()) :: -1

Specified assertion fails for the denoted run.

Wolfgang Schreiner  http://www.risc.uni-linz.ac.at
Extended Static Checking

A step beyond model checking.

- **Application of reasoning calculus**
  Actually also model checking with abstraction does this.
  - Formal specifications in logical framework.
  - Generation of verification conditions.
  - Attempt to automated proofs of these conditions.
    - No assistance from human requested.

- **Constructive subset of program calculus.**
  No additional information required from programmer.
  - Loops handled by finite unrolling.
    - Unrolled program consists of assignments and conditionals only.
  - Simplification actually makes calculus unsound.
    - Not every error is detected.
  - Limited capabilities of underlying prover makes system incomplete.
    - Not every warning really denotes an error.

Infinite state, fully automatic, but falsification rather than verification.
Extended Static Checking

Specifications e.g. formulated in the Java Modeling Language (JML).

```java
/*@ requires a != null && a.length > 0;
   @ assignable \nothing;
   @ ensures (\forall i; 0 <= i && i < a.length; \result <= a[i]);
   @ ensures (\exists int i; 0 <= i && i < a.length; \result == a[i]);
   @*/
public static int main(int[] a)
{
    int n = a.length;
    int m = a[0];
    for (int i=1; i<n; i++)
    {
        if (a[i] < m) m = a[0]; // wrong
    }
    return m;
}
```

Correctness property expressed in a declarative way.
Extended Static Checking

ESC/Java2 (http://secure.ucd.ie/products/opensource/ESCVJava2)

Main.java:16: Warning: Postcondition possibly not established (Post)
   }
  ~

Associated declaration is "Main.java", line 4, col 6:
   @ ensures (\forall\ for all int i; 0 <= i && i < a.length; \result <= a[i]) ...
  ~

Execution trace information:
   Reached top of loop after 0 iterations in "Main.java", line 11, col 4.
   Executed then branch in "Main.java", line 13, col 20.
   Reached top of loop after 1 iteration in "Main.java", line 11, col 4.

Postcondition fails for some run that involves one loop iteration.
1. The RISC Institute

2. Formal Methods at RISC

3. A Short Review of Verification Techniques

4. Computer-Assisted Proving for Program Analysis

5. Conclusions
Computer-Assisted Proving

Use the full version of program calculus.

- **Program calculus in itself is not constructive.**
  Requires additional information from programmer.
  - Reasoning on loops require inductive invariants from external source.
    - Invariant: true before/after every loop iteration.
    - Inductive: strong enough to be propagated across iterations.

- **Program calculus is sound and relatively complete.**
  - Appropriate verification conditions can be automatically generated.

- **Completeness is only relative.**
  - Fully automatic proofs of verification conditions may fail.
    - Theoretical and practical limitations.

- **Computer-assisted interactive proofs have become feasible.**
  Development of the last 10 years or so.
  - Useful SMT (satisfiability modulo theories) checkers have emerged.
    - Automatization of reasoning in certain combinations of basic theories.
    - Employed as building blocks of interactive proving assistants.

Verification depends on user assistance, but only for higher-level tasks.
The RISC ProofNavigator

- **The RISC ProofNavigator**: a proof assistant developed at RISC.
  - Employs the SMT solver CVC Lite (CVCL).
  - Targetted for education in program reasoning.
    - Built after previous experience with other assistants (PVS).
    - Could solve bigger problems with less effort.
- **Focus on practical aspects of proving.**
  Rather than on theoretical elegance.
  - Low-level reasoning completely delegated to SMT solver.
    - Equalities, uninterpreted functions, linear arithmetic, ... 
  - High-level work made as comfortable as possible.
    - Mainly application of pre-selected proof decomposition strategies.
  - Graphical user interface with convenient interaction possibilities.
- **Component of a future program exploration environment.**

The RISC ProgramExplorer


The RISC ProofNavigator

RISC ProofNavigator

Version 1.1 (October 24, 2007)
For the latest version of this program and further information, see http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator.

Copyright (C) 2005, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Author: Wolfgang Schreiner <Wolfgang_Schreiner@risc.uni-linz.ac.at>
The RISC ProofNavigator

ProofNavigator GUI
Eclipse SWT

Output
Input

Proof Tree
Proof State / Declarations
Mozilla

Server
HTTP

Parser ANTLR
RIACA OMlib
DOM

Parser ANTLR
RIACA OMlib
DOM

DOM

Web Browser

XHTML/MathML

OMDoc/OpenMath XML

Declaration/Proof Representation

Declaration/Proof Presentation

ProofNavigator Java

Input

Proof Navigator GUI
Eclipse SWT

Proof Tree
Proof State / Declarations
Mozilla

Output
Input

Server
HTTP

Parser ANTLR
RIACA OMlib
DOM

DOM

Web Browser

XHTML/MathML

OMDoc/OpenMath XML

Declaration/Proof Representation

Declaration/Proof Presentation

ProofNavigator Java

Input

Proof Navigator GUI
Eclipse SWT

Proof Tree
Proof State / Declarations
Mozilla

Output
Input

Server
HTTP

Parser ANTLR
RIACA OMlib
DOM

DOM

Web Browser

XHTML/MathML

OMDoc/OpenMath XML

Declaration/Proof Representation

Declaration/Proof Presentation

ProofNavigator Java

Input
Computer-Assisted Proving

0: \{Input\}
1: \(m := a[0]\)
2: \(i := 1\)
3: \{Invariant\}
4: \textbf{while }i < n \textbf{ do}
5: \textbf{ if } a[i] < m \textbf{ then}
6: \(m := a[i]\)
7: \(i := i + 1\)
8: \{Output\}

execution 0 \rightarrow 1 \rightarrow 2 \rightarrow 3
V1 \equiv Input \land m = a[0] \land i = 1 \Rightarrow Inv(m, i)

execution 3 \rightarrow 4(true) \rightarrow 5(true) \rightarrow 6 \rightarrow 7 \rightarrow 3
V2a \equiv Inv(m, i) \land i < n \land a[i] < m \land m_0 = a[i] \land i_0 = i + 1 \Rightarrow Inv(m_0, i_0)

execution 3 \rightarrow 4(true) \rightarrow 5(false) \rightarrow 7 \rightarrow 3
V2b \equiv Inv(m, i) \land i < n \land a[i] \not< m \land i_0 = i + 1 \Rightarrow Inv(m, i_0)

execution 3 \rightarrow 4(false) \rightarrow 8
V3 \equiv Inv(m, i) \land i \not< n \Rightarrow Output

Verification conditions correspond to program paths.
Computer-Assisted Proving

\[ \text{Input} \equiv n > 0 \land a = olda \land n = oldn \]

\[ \text{Output} \equiv a = olda \land n = oldn \land \\
(\forall i \in \mathbb{N} : i < n \Rightarrow m \leq a[i]) \land \\
(\exists i \in \mathbb{N} : i < n \land m = a[i]) \]

\[ \text{Invariant}(m, i) \equiv \\
n > 0 \land a = olda \land n = oldn \land \\
1 \leq i \leq n \land \\
(\forall j \in \mathbb{N} : j < i \Rightarrow m \leq a[j]) \land \\
(\exists j \in \mathbb{N} : j < i \land m = a[j]) \]

Specification and invariant have to be provided by programmer.
A ProofNavigator Theory

a: ARRAY INT OF INT; olda: ARRAY INT OF INT;
n: INT; oldn: INT; m: INT; m_0: INT; i: INT; i_0: INT;

Input: BOOLEAN =
a = olda AND n = oldn AND n > 0;

Output: BOOLEAN =
a = olda AND n = oldn AND n > 0 AND
(FORALL(i: INT): 0 <= i AND i < n => m <= a[i]) AND
(EXISTS(i: INT): 0 <= i AND i < n AND m = a[i]);

Invariant: (INT, INT) -> BOOLEAN =
LAMBDA(m: INT, i: INT):
a = olda AND n = oldn AND n > 0 AND 1 <= i AND i <= n AND
(FORALL(j: INT): 0 <= j AND j < i => m <= a[j]) AND
(EXISTS(j: INT): 0 <= j AND j < i AND m = a[j]);

V1: FORMULA
Input AND m = a[0] AND i = 1 => Invariant(m, i);

V2_a: FORMULA
Invariant(m, i) AND i < n AND a[i] < m AND m_0 = a[i] AND i_0 = i+1 =>
Invariant(m_0, i_0);

V2_b: FORMULA
Invariant(m, i) AND i < n AND NOT(a[i] < m) AND i_0 = i+1 =>
Invariant(m, i_0);

V3: FORMULA
Invariant(m, i) AND NOT(i < n) => Output;
The RISC ProofNavigator

Proof Tree:

Invariants:
- \( a = \text{odd} \land n = \text{odd} \land a > 0 \)
- \( a = \text{odd} \land n = \text{odd} \land a > 0 \) \( \land (\forall i \in \mathbb{N}: j < n \land m = o[i]) \)
- \( \forall i \in \mathbb{N}: j < n \land m = o[i] \)

Variables:
- \( a \in \mathbb{Z}, i \in \mathbb{N}: a = \text{odd} \land n = \text{odd} \land a > 0 \land 1 \leq i \land i \leq n \)
- \( \forall i \in \mathbb{N}: j < n \land m = o[i] \)

Formulas:
- \( V1 = \text{Input} \land \text{if} \land \text{if} \land \text{Invariant}(m, i) \)
- \( V2 = \text{Invariant}(m, i) \land i < n \land o[i] < m \land m_0 = o[i] \land i_0 = i + 1 \rightarrow \text{Invariant}(m, i_0) \)
- \( V3 = \text{Invariant}(m, i) \land i < n \rightarrow \text{Output} \)

Input/Output:
- Formula V2_a
- Proof read (proof status: trusted, closed, absolute)
- Formula V2_b
- Proof read (proof status: trusted, closed, absolute)
- Formula V3
- Proof read (proof status: trusted, closed, absolute)
- Formula V2A0
- Proof read (proof status: untrusted)
- File minimum.txt read.
The RISC ProofNavigator
The RISC ProofNavigator

V1:  
  • [2hg]: expand Input, Invariant
  • [6ko]: scatter
    [21d]: proved (CVCL)
    [31d]: proved (CVCL)
  • [41d]: auto
    [nej]: proved (CVCL)

V2a:  
  • [2hg]: expand Input, Invariant
  • [6ko]: scatter
    [21d]: proved (CVCL)
    [31d]: proved (CVCL)
  • [41d]: auto
    [nej]: proved (CVCL)

V2b:  
  • [hqk]: expand Invariant
  • [thu]: scatter
    [hfa]: proved (CVCL)
    [ifa]: proved (CVCL)
  • [jfa]: auto
    [b41]: proved (CVCL)
  • [kfa]: auto
    [i3q]: proved (CVCL)

V3:  
  • [4hg]: expand Invariant, Output
  • [nx5]: scatter
    [4pd]: proved (CVCL)
  • [5pd]: auto
    [udv]: proved (CVCL)

Expanding definitions, decomposing proofs, instantiating quantifiers.
Dealing with Proof Failures

In practice, the real problem is dealing with proof failures appropriately.

- **Program is wrong** (actually, does not satisfy specification).
  Perhaps specification does not express intended meaning.

- Failed verification condition represents an erroneous program path.

- **Loop/system invariant is too strong.**
  - Not preserved by loop iteration/some system transition.

- **Loop/system invariant is too weak.**
  - In particular, not inductive: cannot be propagated through loop iteration/all system transitions.

- **Proof strategy was inadequate.**
  - Inadequate instantiation of quantifiers.
  - Incomplete reasoning of SMT solvers.
  - Insufficient auxiliary knowledge (lemmas).

It requires training to recognize the reason of a failure and deal with it.
A Wrong Program

0: \{Input\}
1: \( m := a[0] \)
2: \( i := 1 \)
3: \{Inv\}
4: while \( i < n \) do
5: \quad if \( a[i] < m \) then
6: \quad \quad \( m := a[0] \) // wrong
7: \quad \quad \( i := i + 1 \)
8: \{Output\}

path 3 \rightarrow 4(true) \rightarrow 5(true) \rightarrow 6 \rightarrow 7 \rightarrow 3
\( V2a \equiv Inv(m, i) \land i < n \land a[i] < m \land \overline{m_0 = a[0]} \land i_0 = i + 1 \Rightarrow Inv(m_0, i_0) \)

Failure in \( V2a \) (3 \rightarrow 4(true) \rightarrow 5(true) \rightarrow 6 \rightarrow 7 \rightarrow 3\).
A Wrong Program

**Formula [Wrong] proof state [vyr] (autosimp)**

Constants (with types): Input, \( m_0, a, n, j_0 \), Output, \( m, \text{oldn}, j_1, i, i_0, \text{olda}, \text{Invariant} \).

- \( \text{ed2} \): \( \text{olda} = a \)
- \( \text{o5v} \): \( \text{oldn} = n \)
- \( \text{1ha} \): \( j_0 < i \)
- \( \text{srh} \): \( m = a[j_0] \)
- \( \text{tfb} \): \( \forall j \in \mathbb{N}: j < i \Rightarrow m \leq a[j] \)
- \( \text{ysm} \): \( m_0 = a[0] \)
- \( \text{kpd} \): \( i_0 = 1 + i \)
- \( \text{l6h} \): \( i < n \)
- \( \text{l41} \): \( a[i] < m \)
- \( \text{b2x} \): \( j_1 < 1 + i \)
- \( \text{w25} \): \( m_0 \leq a[j_1] \)

**Parent:** [d23]

Assignment \( m_0 = a[0] \) does not preserve invariant \( m_0 \leq a[i] \).
Invariant Too Strong

0: \{Input\}
1: \( m := a[0] \)
2: \( i := 1 \)
3: \{Inv\}
4: \texttt{while } i < n \texttt{ do}
5: \quad \texttt{if } a[i] < m \texttt{ then}
6: \quad \quad m := a[i]
7: \quad \quad i := i + 1
8: \{Output\}

\( \text{Inv}(m, i) \equiv \)
\[
\begin{align*}
  n > 0 & \land a = olda & \land n = oldn & \land \\
  1 & \leq i \leq n & // & \text{too strong} \\
  (\forall j \in \mathbb{N} : j < i \Rightarrow m & \leq a[j]) & \land \\
  (\exists j \in \mathbb{N} : j < i & \land m = a[j])
\end{align*}
\]

path \( 3 \to 4(\text{true}) \to 5(\text{true}) \to 6 \to 7 \to 3 \)
\( V2a \equiv \text{Inv}(m, i) \land i < n \land a[i] < m \land m_0 = a[i] \land i_0 = i + 1 \Rightarrow \text{Inv}(m_0, i_0) \)

Failure in \( V2a \ (3 \to 4(\text{true}) \to 5(\text{true}) \to 6 \to 7 \to 3) \) and in \( V2b \).
Invariant Too Strong

Invariant $1 + i < n$ is not preserved.
Invariant Too Weak

0: \{Input\}
1: \text{m := a[0]}
2: \text{i := 1}
3: \{Inv\}
4: while \text{i < n do}
5: \quad \text{if a[i] < m then}
6: \quad \quad \text{m := a[i]}
7: \quad \quad \text{i := i + 1}
8: \{Output\}

\text{Inv(m, i) \equiv}
\quad n > 0 \land a = \text{old}a \land n = \text{old}n \land
\quad // too weak: 1 \leq i \leq n missing
\quad (\forall j \in \mathbb{N} : j < i \Rightarrow m \leq a[j]) \land
\quad (\exists j \in \mathbb{N} : j < i \land m = a[j])

path 3 \rightarrow 4(\text{false}) \rightarrow 8
\text{V3 \equiv Inv(m, i) \land i \not< n \Rightarrow Output}

Failure in V3 (path 3 \rightarrow 4(\text{false}) \rightarrow 8).
Invariant Too Weak

Formula [TooWeak] proof state [dtw] (autosimp)

Constants (with types): \( m_0 \), Input, WInvariant, \( a, n, j_0 \), Output, oldn, \( m, i, i_0 \), olda, Invariant, SInvariant.

- \texttt{ed2}: \( \text{olda} = a \)
- \texttt{o5v}: \( \text{oldn} = n \)
- \texttt{ybv}: \( 0 \leq j_0 \)
- \texttt{1ha}: \( j_0 < i \)
- \texttt{srh}: \( m = a[j_0] \)
- \texttt{v5o}: \( \forall j \in \mathbb{Z}: 0 \leq j \land j < i \Rightarrow m \leq a[j] \)
- \texttt{cu2}: \( 0 < n \)
- \texttt{c1r}: \( \forall i \in \mathbb{Z}: m = a[i] \Rightarrow 0 > i \lor i \geq n \)
- \texttt{6ha}: \( j_0 < n \)

Parent: \texttt{[t3o]}

Invariant lacks \( i \leq n \).
Other Reasoning Tasks

Open question: is the specification adequate?

- Do given input/output samples satisfy the specification?
  \[ \text{Input}(x_1) \land \text{Output}(x_1, y_1) \]
  \[ \text{Input}(x_2) \land \text{Output}(x_2, y_2) \]
  \[ \ldots \]

- Is the specification not trivial?
  \[ \exists x, y : \text{Input}(x) \land \neg \text{Output}(x, y) \]

- Can the specification be satisfied by any implementation?
  \[ \forall x : \text{Input}(x) \Rightarrow \exists y : \text{Output}(x, y) \]

Before verification, reasoning should be applied to validate specifications.
Analyzing Concurrent Systems

Server:
  local given, waiting, sender
begin
  given := 0
  for i in 1..N do waiting[i] := 0
  loop
    sender := receiveRequest()
    if sender = given then
      if forall i in 1..N: waiting[i] = 0 then
        given := 0
      else
        choose i in 1..N: waiting[i] = 1
        given := i; waiting[i] := 0
        sendAnswer(given)
    endif
    elseif given = 0 then
      given := sender
      sendAnswer(given)
    else
      waiting[sender] := 1
    endif
  endloop
end Server

Client(ident):
  param ident
begin
  loop
    ...
    sendRequest()
    receiveAnswer()
    ...
    // critical region
    sendRequest()
  endloop
end Client

A system of $N$ clients linked via $2N$ buffered channels to a server that maintains mutual exclusion of shared resource access among the clients.
Analyzing Concurrent Systems

Basic strategy: again invariant reasoning.

- **System** = \langle State, Init, Trans \rangle
  
  Init \subseteq State
  Init(x, \ldots) \equiv \ldots
  
  Trans \subseteq State \times State
  Trans(\langle x, \ldots \rangle, \langle x', \ldots \rangle) \equiv
  T_1(\langle x, \ldots \rangle, \langle x', \ldots \rangle) \lor \ldots \lor T_n(\langle x, \ldots \rangle, \langle x', \ldots \rangle)

- Verification Goal: **System \models □P**
  
  P \subseteq State, P(x, \ldots) \equiv \ldots
  Inv \subseteq State, Inv(x, \ldots) \equiv \ldots
  
  VA: Inv(x, \ldots) \Rightarrow P(x, \ldots)
  VB: Init(x, \ldots) \Rightarrow Inv(x, \ldots)
  VC1: Inv(x, \ldots) \land T_1(\langle x, \ldots \rangle, \langle x', \ldots \rangle) \Rightarrow Inv(x', \ldots)
  
  \ldots
  VCn: Inv(x, \ldots) \land T_n(\langle x, \ldots \rangle, \langle x', \ldots \rangle) \Rightarrow Inv(x', \ldots)

To prove safety of state property **P** in **System**, prove that every transition of **System** preserves invariance of some stronger property **Inv**.
Analyzing the Client/Server System

Inv1 ≡

Initial
⇒

Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer)

Inv2 ≡

Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer)

^ Next
⇒

Invariant(pc0, request0, answer0, given0, waiting0, sender0, rbuffer0, sbuffer0)

MutEx ≡

Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer)
⇒

¬ (∃i ∈ Index, j ∈ Index: i ≠ j ∧ pc[i] = C ∧ pc[j] = C)

Main work is developing the correct system invariant.
The System Invariant

FORALL (i: Index):
   (pc[i] = R =>
     (sbuffer[i] <=> FALSE) AND (answer[i] <=> FALSE) AND
     (i /= given => (request[i] <=> FALSE) AND (rbuffer[i] <=> FALSE) AND sender /= i) AND
     (i = given => (request[i] <=> TRUE) OR (rbuffer[i] <=> TRUE) OR sender = i) AND
     ((request[i] <=> FALSE) OR (rbuffer[i] <=> FALSE))) AND
   (pc[i] = S =>
     ((sbuffer[i] <=> TRUE) OR (answer[i] <=> TRUE) =>
     (request[i] <=> FALSE) AND (rbuffer[i] <=> FALSE) AND sender /= i) AND
     (i /= given => (request[i] <=> FALSE) OR (rbuffer[i] <=> FALSE)) AND
   (pc[i] = C =>
     (request[i] <=> FALSE) AND (rbuffer[i] <=> FALSE) AND sender /= i AND
     (sbuffer[i] <=> FALSE) AND (answer[i] <=> FALSE)) AND
   (pc[i] = C OR (sbuffer[i] <=> TRUE) OR (answer[i] <=> TRUE) =>
     given = i AND
     (FORALL (j: Index): j /= i =>
   (sender = 0 AND ((request[i] <=> TRUE) OR (rbuffer[i] <=> TRUE)) =>
     (sbuffer[i] <=> FALSE) AND (answer[i] <=> FALSE)) AND
   (sender = i =>
     (sender = given AND pc[i] = R =>
       (request[i] <=> FALSE) AND (rbuffer[i] <=> FALSE)) AND NOT waiting[i] AND
     (pc[i] = S AND i /= given => (request[i] <=> FALSE) AND (rbuffer[i] <=> FALSE)) AND
     (pc[i] = S AND i = given => (request[i] <=> FALSE) OR (rbuffer[i] <=> FALSE))) AND
   ...

Systematic initial construction and subsequent refinement during proof.
Proving Mutual Exclusion (VA and VB)

[z3f]: expand Invariant, IC, IS
[hnj]: scatter
[znj]: auto
[nju]: proved (CVCL)

[oas]: expand Initial, Invariant, IC, IS
[eij]: scatter
[5ul]: auto
[uvj]: proved (CVCL)
[6ul]: auto
[2u6]: proved (CVCL)
[avl]: auto
[cuv]: proved (CVCL)
[bvl]: auto
[jt1]: proved (CVCL)
[ct1]: auto
[qsb]: proved (CVCL)
[dvl]: auto
[xrr]: proved (CVCL)
[evl]: auto
[5qn]: proved (CVCL)
[fvl]: auto
[fqd]: proved (CVCL)
[gy1]: auto
[mpz]: proved (CVCL)
[hv1]: proved (CVCL)
[h5h]: auto
[p3z]: proved (CVCL)
[i5h]: auto
[gjb]: proved (CVCL)
...

...[q5h]: proved (CVCL)
[q5i]: proved (CVCL)
[r5i]: proved (CVCL)
[s5i]: proved (CVCL)
[t5i]: proved (CVCL)
[u5i]: auto
[1br]: proved (CVCL)
[v5i]: auto
[roy]: proved (CVCL)
[w5i]: auto
[i56]: proved (CVCL)
[x5i]: proved (CVCL)
[y5i]: auto
[uwo]: proved (CVCL)
[z5i]: auto
[nbw]: proved (CVCL)
[z5j]: auto
[nbn]: proved (CVCL)
[15j]: auto
[eou]: proved (CVCL)
[25j]: proved (CVCL)
[35j]: proved (CVCL)
[45j]: proved (CVCL)
[55j]: proved (CVCL)
[65j]: proved (CVCL)

Single application of command autostar.
Proving Invariance (VC1-VC10)

| [pas]: scatter                                 | [st6]: scatter                                  | [h4b]: scatter                                 |
| [1bh]: expand Next                             | [aef]: expand Invariant                          | [tob]: expand Invariant                         |
| [pzi]: split bfv                               | [cwk]: scatter                                  | [h1g]: scatter                                 |
| [1eh]: decompose                               | [q6]: auto                                      | [t4]: auto                                      |
| [pkr]: expand RS                               | [seg]: proved (CVCL) ... (21 times)              | [hpk]: proved (CVCL) ... (36 times)             |
| [1pn]: split 5xv                               | [w16]: proved (CVCL)[neh]: scatter ... (12 times) |                                            |
| [pt6]: expand Invariant                        | [4oc]: expand RC                                 |                                            |
| [1cw]: scatter                                 | [tt6]: scatter                                  | [nuh]: split nwz                                |
| [puh]: auto                                    | [hp6]: expand Invariant                          |                                            |
| [143]: proved (CVCL) ... (20 times)            | [tw1]: scatter                                  |                                            |
| [tuh]: proved (CVCL) ... (15 times)            | [hq5]: auto                                      |                                            |
| [qt6]: expand Invariant                        | [tbj]: proved (CVCL) ... (27 times)              |                                            |
| [sqn]: scatter                                 | [nqv]: proved (CVCL) ... (6 times)               |                                            |
| [avi]: auto                                    |                                            |                                            |
| [cct]: proved (CVCL)[meh]: scatter ... (26 times) |                                            |                                            |
| [w3z]: expand External                         |                                            |                                            |
| [gvi]: proved (CVCL) ... (6 times)             |                                            |                                            |
| [3rk]: split 1he                               |                                            |                                            |
| [g4b]: scatter                                 |                                            |                                            |
| [mdh]: expand Invariant                        |                                            |                                            |
| [zyk]: expand Invariant                        |                                            |                                            |
| [rvj]: scatter                                 |                                            |                                            |
| [zgj]: auto                                    |                                            |                                            |
| [rhd]: proved (CVCL) ... (31 times)            |                                            |                                            |
| [2f3]: proved (CVCL) ... (1 times)             |                                            |                                            |

Ten subproofs each requiring only single application of autostar.

Wolfgang Schreiner

http://www.risc.uni-linz.ac.at
1. The RISC Institute

2. Formal Methods at RISC

3. A Short Review of Verification Techniques

4. Computer-Assisted Proving for Program Analysis

5. Conclusions
Applicability of Verification Technologies

To which extent is all of this applicable in practice?

- **Simple safety properties**: real programs.
  Prevent program crashes at runtime.

- **Software model checking**, extended static checking.

- **Simple protocol properties**: abstract models, partially programs.
  Protocol properties expressible by regular expressions.

- **LTL/CTL model checking**, software model checking.

- **Precondition checking**: real programs.
  Ensure internal program consistency.

- **Extended static checking** (limited), computer-assisted proving.

- **General correctness**: abstract models, critical program parts.
  Protocol properties beyond regular expressions, postconditions, adequacy of specifications.

- **Computer-assisted proving**.

  For complex properties, work on designs rather than on implementations.
Current Formal Methods Research

Mainly aims at hiding reasoning from programmers.

- A lot of reasoning technology is hidden inside tools.
  - Model Checker BLAST: uses SMT solvers.
    - CEGAR principle: automated construction of (simple) invariants.
    - No declarative specifications, no invariants, no reasoning assistance.
  - Extended Static Checker ESC/Java2.
    - Generates verification conditions and discharges them by SMT solvers.
    - Declarative specifications, no invariants, no reasoning assistance.
  - Program verifier Spec# (Microsoft).
    - Declarative specifications, invariants, no reasoning assistance.

- Also academic program verifiers focus on automation.
  - KeY (Karlsruhe, Koblenz, Chalmers)
    - Interactive proving assistant for Java programs.
    - Many program proofs work fully automatic.
    - Weak interface for understanding and assisting reasoning tasks.

Little support for reasoning as an explicit part of software development.
Teaching Formal Methods

Experience from JKU/FH courses on “FM in Software Development”.

- **Learning to use the language of logic as a specification language.**
  Successful verifications may be meaningless.
- Classical predicate logic as well as temporal logic.
- How to express intuitive ideas in a formally precise way.
- How to ensure that a formal specification is adequate.

- **Learning to interpret/deal with failed verifications.**
  What does it mean, if the verification does not run through?
- For model checkers (counterexample traces) as well as for extended static checkers and provers (failed verification conditions).
- Dealing with uncertainty (proof strategy, invariant).

- **Learning to understand/develop/refine invariants.**
  Dealing with large/infinite state verifications.
- Loop invariants and concurrent system invariants.
- There is no practical substitute for manual invariant construction.

Verifying correct programs (given the invariants) is comparatively easy.
Teaching Formal Methods

- But learning the language of logic is so difficult!
  - Yes, like learning a programming language is.
  - One needs training in both.
- But formal reasoning is so difficult!
  - Yes, like writing correct programs is.
  - One needs training in both.
- Programmers should describe their artefacts and argue about them.
  - There is a general inability of communicating about programs.
  - Formal specification and reasoning is just the core essence.
- Teaching should put emphasis on correctness (also in the small).
  - We teach students to “engineer” large applications.
  - But we don’t teach them the patience to get the details right.
    - Many programs written (also by professionals) are a shame.
- Computer-assisted reasoning tools may support/enforce this.
  - Like a compiler supports/enforces correct syntax and typing.

Reasoning tools assist (not replace) education on writing correct software.
The RISC ProgramExplorer

My own contribution to the field; currently under development.

Wolfgang Schreiner http://www.risc.uni-linz.ac.at