Verifying Java-KE programs

- Verifying virtual methods / interface properties
- Verifying heap-manipulating object-oriented programs

Example: Virtual methods

```
interface MyInf {
  int m( int par );
}

class Impl1 implements MyInf {
  int m( int par ) {
    if( p < 0  ) res = 0; else res = p;
  }
}

class Impl2 implements MyInf {
  int m( int par ) { res = 3; }
}

public class VerifVirtualMethods {
  int main( MyInf par ) {  res = par.m( -9 ); }
}
```

Example: Virtual methods (2)

Prove:

$$\vdash \{\ \mathit{par} \neq \mathit{null}\ \}\ \texttt{VerifVirtualMethods@main}\ \{\ \mathit{res} \geq 0\ \}$$

Example: Heap-manipulating OO programs

```
class Null {}

class Node {
  int elem;
  Node next;
}

class List {
  Node elems;

  boolean isempty() { res = (elems == null); }

  void add( int par ) { ... } // next slide

  void append( List par ) { ... } // second next slide
}
```

Example: Heap-manipulating OO programs (2)

```
void add( int par ) {
  Node newNd;
  Node oldNd;
  newNd = new Node();
  newNd.elem = par;
  oldNd = this.elems;
  newNd.next = oldNd;
  this.elems = newNd;
}
```

Example: Heap-manipulating OO programs (3)

```
void append( List par ) {
  Node appl;
  appl = par.elems;
  while( appl != null ) {
    int el;
    el = appl.elem;
    this.add( el );
    appl = appl.next;
  }
}
```

Example: Heap-manipulating OO programs (4)

Develop:

1. abstraction predicates for lists
2. invariant of object store/heap
3. specifications for `isempty`, `add` and `append`

Prove correctness of class `List` (see lecture and technical report)

Section 8.7

Software verification tools

Overview

Considered techniques and tools:

- Tools for interactive software verification (e.g., KeY, Why)
- Extended static checking (e.g., Spec#, Chalice, VeriFast, BLAST)
- Refinement and compatibility checking (e.g., Event B, KIV, BCVerifier)
- Automated theorem proving, in particular:
  - Superposition provers (e.g., SPASS, E)
  - SMT solvers and model checkers (e.g., Z3, SPIN)

Subsection 8.7.1

Tools for interactive software verification

General approach

- Programming-language-specific front end/development environment
- Programming-language-specific specification language
- Verification condition generator (VCG)
- Possibly several provers to discharge the VCs (automated and/or interactive)

Example systems

- KeY: www.key-project.org/
  - programming language: JavaCard; specifications in JML
  - based on an interactive prover for dynamic logic
  - makes extensive use of symbolic evaluation
- Why, Why3: why.lri.fr/
  - programming languages: Java subset, C subset
  - specific specification languages
  - general-purpose verification condition generator
  - uses many interactive and automated provers

Discussion of refinement-based approach

- Advantages:
  - quite close to programming language and programmer
  - special IDEs for programs, specifications and proofs
  - can smoothly integrate powerful logics and dedicated automated techniques
- Disadvantages:
  - expensive solution
  - not very flexible w.r.t. extensions
  - meta-logical aspects cannot be handled

Subsection 8.7.2

Extended static checking

General approach

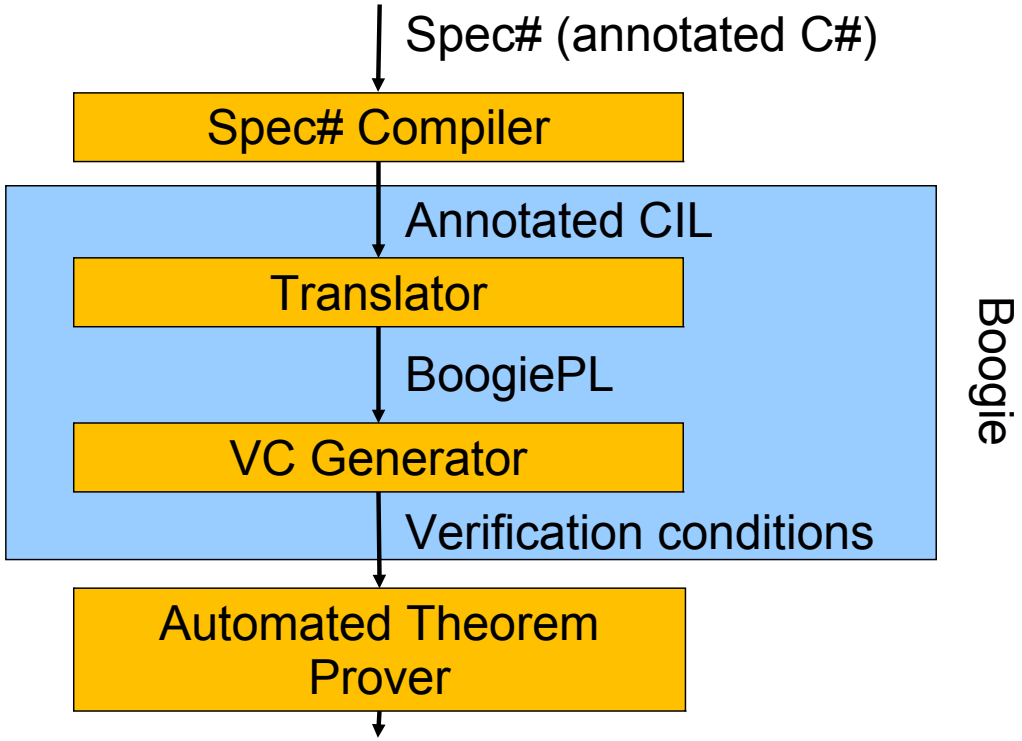
- Programming-language-specific front end/development environment
- Programming-language-specific specification language
- Verification condition generator (VCG)
- Automated prover to discharge the VCs

Example systems

- Spec#: research.microsoft.com/en-us/projects/specsharp/
  - programming and specification language: Spec# (extension of C#)
  - specific focus on modularity of specifications
  - uses first-order ATP
  - also supports dynamic checking
- VeriFast: people.cs.kuleuven.be/~bart.jacobs/verifast/
  - verifier for single-threaded and multi-threaded C and Java programs
  - pre- and postconditions written in separation logic
  - user guides the proofs by so-called “lemma functions”
  - uses the SMT solver Z3
- BLAST: mtc.epfl.ch/software-tools/blast/
  - software model checker for C programs
  - checking temporal safety properties
  - uses **C**ounter**E**xample-**G**uided automatic **A**bstraction **R**efinement
  - succeeds or provides a counterexample or fails

Typical architecture for ESC

Spec# tool architecture:



Discussion of extended static checking

- Advantages:
  - close to programming language and programmer
  - good integration with normal IDEs
  - in principle, no contact with the prover needed
- Disadvantages:
  - specifications less expressive (why?), in particular w.r.t. abstraction
  - error messages can be tricky if checking fails
  - helping the prover can get difficult

Subsection 8.7.3

Refinement and compatibility checking

General approach

- Support the formal development from software models to programs
- Refinement relates software models on different levels of abstraction
- Compatibility checking relates different program versions
- Proofs based on simulation techniques
- Possibly several provers to discharge the VCs (automated and/or interactive)

Example systems refinement

- Event B: www.event-b.org/
  - correctness by construction in the tradition of VDM
  - system = software + environment: represented as transition systems
  - B notation following the Z notation
  - specific development and proof platform Rodin
  - programs are generated from most concrete model
- KIV: www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/
  - formal systems development and interactive verification
  - specification support:
    - functional aspects: abstract data types and HOL
    - state-based aspects: programs and abstract state machines
  - supports various kinds of refinements
  - sophisticated IDE for proof engineering

Example systems compatibility checking

BCVerifier: softech.informatik.uni-kl.de/bcverifier/

- checks two packages written in a Java subset for backward compatibility
- supports a specification language for writing coupling invariants
- uses Boogie as checking platform

Discussion

- Software verification goes beyond program verification
- Other interesting aspects:
  - Correctness of software systems
  - Correctness of refactoring methods
  - Correctness of compilers and programming tools

Subsection 8.7.4

ATP: Automated theorem proving

Techniques for automated verification

A rough classification:

The software verification tools use many techniques for automating proofs or proof steps, in particular:

- Superposition provers (e.g., SPASS, E)
- SMT solvers and model checkers (e.g., Z3, SPIN)
- Abstract interpretation and abstraction refinement