Formal Methods

Software Engineering 2008
Bernd Schoeller
ETH Zurich
The ideal of correct software has long been the goal of research in Computer Science. We now have a good theoretical understanding of how to describe what programs do, how they do it, and why they work.

This understanding has already been applied to the design, development and manual verification of simple programs of moderate size that are used in critical applications. Automatic verification could greatly extend the benefits of this technology.

[...] the time is ripe to embark on an international Grand Challenge project to construct a program verifier that would use logical proof to give an automatic check of the correctness of programs submitted to it.

(Hoare and Misra, July 2005)
Lecture Contents

- Software verification overview
  - Bottom-up verification
  - Top-down verification
- Specification problems for verification
  - Specify what software does
  - Specify what software does **not**
- Tool demonstration
  - Rodin, Slam, (Spec#)
Verification Problems

- State space explosion
- Combinatorial explosion
- Overall complexity
- Specification problems
Implementation vs. Specification

**Implementation**
- Concrete
- Constructive
- Deterministic (mostly)
- All is finite
- Describes everything
- Can be executed by a machine

**Specification**
- Abstract
- Descriptive
- Non-deterministic
- Can describe infinity
- Describes only parts
- Can not be executed by a machine
Information Hiding

- Interface A
  - Implementation A

- Interface B
  - Implementation B

- Interface C
  - Implementation C

- Interface D
  - Implementation D
Modular verification

<table>
<thead>
<tr>
<th>Interface A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation A</td>
</tr>
</tbody>
</table>
Modular verification

<table>
<thead>
<tr>
<th>Interface B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation B</td>
</tr>
</tbody>
</table>
Modular verification

- Interface C
  - Implementation C
  - Interface A
  - Interface B
Modular verification

- Interface D
- Implementation D
- Interface C
- Interface B
Assumption of System Correctness

If each implementation is correct with respect to its interface, based on the interfaces of the modules it uses, then the overall system is correct.
Verification Technology

• Axiomatic Semantics
  – Hoare triples: \{P\} S \{Q\}
  – Weakest precondition computation

• Provers
  – Automatic (Simplify, Z3)
  – Interactive (Isabelle, COQ)
  – Model checking (blast, slam, ...)

The ideal specification

too strong
- implementation inefficient / impossible
- difficult to change (information hiding)

too weak
- reasoning difficult for clients
- not composable
- side-effects: proofs impossible
Specification problems of Design by Contract

Contracts on unbounded data

Possible Solution: model-based contracts

Frame problem

Possible Solution: Dynamic Frame Contracts (DFC)
class LINKED_QUEUE[G] feature

  put (v: G)
     -- Put `v' into the queue.
     ensure
       ?
  remove
     -- Remove the oldest element from the queue.
     ensure
       ?

  item: G
     -- Oldest element in the queue
     ensure
       ?

end
Model-based contracts

class LINKED_QUEUE[G] feature

put (v: G)
   -- Put `v' into the queue.
   ensure
      model = old model.prepended (v)

remove
   -- Remove the oldest element from the queue.
   ensure
      model = old model.front

item: G
   -- Oldest element in the queue
   ensure
      Result = model.last

model: MML_SEQUENCE[G]
   -- Contents of queue viewed as a sequence
end
**Frame Problem**

- You have a green box in Zürich and you want to have a red box in Luzern.

<table>
<thead>
<tr>
<th>Move Boxes Inc</th>
<th>All-red Painters AG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specification</strong>: Give us the box and some money in Zürich, and we will transport the box to Luzern within 2 days.</td>
<td><strong>Specification</strong>: Give the box and some money, and we will change the color of the box to red.</td>
</tr>
</tbody>
</table>
Frame Problem and DbC

class PERSON feature
  -- Interface view
  last_name: STRING
  set_last_name (name: STRING)
    ensure
      last_name = name
end

class CIVIL_REGISTRY_OFFICE feature
  marry (first_person, second_person: PERSON)
    require
      first_person /= second_person
    do
      second_person.set_last_name (first_person.last_name)
    ensure
      first_person.last_name = second_person.last_name
end
Frames

SECOND_PERSON. set_last_name (command)  

FRAME A ("Set of Resources")

modify ("write effect")

FRAME B ("Set of Resources")

first_person. last_name (query)  

use ("read effect")

Resource can be: memory (fields), file, network state, ...
Hidden dependencies

PERSON A

first_person

PERSON B

second_person
Frame specifications

**strong specification**

class PERSON feature
   -- Interface view
   last_name: STRING
       use
       { Current }

   set_last_name (name: STRING)
       ensure
       last_name = name
   modify
       { Current }

end

**weak specification**

class PERSON feature
   -- Interface view
   last_name: STRING
       use
       personal_data

   set_last_name (name: STRING)
       ensure
       last_name = name
   modify
       personal_data
       { Current }

end
Video Ballet Verifier
Top-Down Verification

- Specification
- 2. Specification
- 3. Specification
- 4. Specification
- Implementation
B Method

- Development by stepwise refinement
- Invented by J. R. Abrial (currently giving lectures at ETH!)
- Provers: Atelier B, B4Free, Rodin Platform
- Animator and Model Checker: ProB
- Industrial application:
  - Embedded devices
  - > 100000 LOC verified for the Paris Metro
Example: Traffic Light

driveNS: BOOL
driveEW: BOOL

INVARIANT: not (driveNS and driveEW)

EVENT: goNS := WHERE driveNS = FALSE
THEN driveEW := TRUE END
Example: Traffic Light

redLightNS,yellowLightNS,greenLightNS: BOOL
redLightEW,yellowLightEW,greenLightEW: BOOL

INV: greenLightNS = driveNS, greenLightEW = driveEW

EVENT: goEW := WHERE redLightNS = FALSE THEN
redLightEW, yellowLightEW, greenLightEW := FALSE,FALSE,TRUE END
Prove Obligations

• Within a machine, you have to prove:
  – That the initialization establishes the invariant
  – That each event re-establishes the invariant

• Within a refinement, you also have to prove
  – That the new initialization implies the old initialization
  – That refined events imply the old state change
  – That the new events refine SKIP
Demo Rodin Workbench
Problems with Top-Down Verification

- It is a mind-twister
- One needs to know all specifications from very early on
- Composition is difficult
- Reuse is very difficult
- Uncommon thinking
- Many common SE practice cannot be easily transferred: Information hiding, OO, pointers, ...

But they are working on it, so keep your eyes open