Semantic Inspection of Software Artifacts

Tim Heyer

Objective:
Tools and practical means for systematic software (code and design) inspection
Projects

- Verification Automation in Software Development
  - Supported by NUTEK competence center ISIS
  - Started in Nov 1995, ended in Oct 1997
  - ABB Industrial Systems and RTSLAB, Linköping University

- Tool support for design inspection
  - Supported by NUTEK
  - Started in Apr 1998
  - Ericsson SoftLab AB and TCSLAB, Linköping University
## Testing, Review, and Formal Verification

<table>
<thead>
<tr>
<th>Testing</th>
<th>Reviews/Inspections</th>
<th>Formal verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>• very late</td>
<td>• earlier</td>
<td>• earlier</td>
</tr>
<tr>
<td></td>
<td>• “easy”</td>
<td>• formal proof</td>
</tr>
<tr>
<td></td>
<td>• focus on style</td>
<td>• “difficult”</td>
</tr>
<tr>
<td></td>
<td>• unsystematic</td>
<td></td>
</tr>
</tbody>
</table>
Hoare’s Partial Correctness Assertion Method

- Assertions and code
- Generation of verification conditions
- Formal proof
- Follow-up
Example: Hoare’s Method

\[
\{ x=x@\text{pre} \land y=y@\text{pre} \} \ x=x+y; \ y=x-y; \ x=x-y; \ \{ x=y@\text{pre} \land y=x@\text{pre} \}
\]

Proof outline:
\[
\{ x=x@\text{pre} \land y=y@\text{pre} \}
\{(x+y)-((x+y)-y)=y@\text{pre} \land ((x+y)-y)=x@\text{pre} \}
\]
\[
x=x+y;
\{ x-(x-y)=y@\text{pre} \land x-y=x@\text{pre} \}
\]
\[
y=x-y;
\{ x-y=y@\text{pre} \land y=x@\text{pre} \}
\]
\[
x=x-y;
\{ x=y@\text{pre} \land y=x@\text{pre} \}
\]

Verification condition:
\[
x=x@\text{pre} \land y=y@\text{pre} \Rightarrow (x+y)-((x+y)-y)=y@\text{pre} \land ((x+y)-y)=x@\text{pre}
\]
Our Approach

- Annotated artifact
- Automatic generation of questions
- Inspection (answering the questions)
- Follow-up
Annotations and Questions

Assertions

• Predicate logic

• Predicates without formal definition
  ◦ Easier to state
  ◦ Easier to reason about

Questions

• Analytic implication

Assume:

\[
\text{sorted}(a, 1, n) \\
\text{a}[n+1] \geq \text{max}(a, 1, n)
\]

Then:

\[
\text{sorted}(a, 1, n+1)
\]
Our Approach to Code Verification (Java)

1. Detailed design and code
2. Automatic generation of questions
3. Code inspection (answering the questions)
4. Follow-up
Detailed Design

Annotated class descriptions

- Class invariants
- Method pre- and postconditions
- Constants and lists of modified variables

```java
public void qsort(int[] v, /*+const*/ int l, /*+const*/ int r)
/**+
   requires
   true;
   ensures
   sorted(v@pre, v, l@pre, r@pre);
   modifies
   v;
*/
```
Implementation

Annotated code

- Java subset
- Intermediate assertions, loop invariants

WrapInt i=new WrapInt(0);
if (l < r)
{
    split(v, l, r, i);
    /*+
    ensures
    partition(v@pre, v,
    l@pre, r@pre, i) ;
    */
    qsort(v, l, i.val()−1);
    qsort(v, i.val()+1, r);
}
Question Generation

- Axiomatic semantics
- Questions for:
  - Preconditions
  - Postconditions and class invariants
  - Loop invariants
  - Inheritance

Assume:
\[ l@pre < r@pre \]
\[ \text{partition}(v@pre, v\#3, l@pre, r@pre, i\#1) \]
\[ \text{sorted}(v\#3, v\#2, l@pre, i\#1-1) \]
\[ \text{sorted}(v\#2, v\#1, i\#1+1, r@pre) \]

Then:
\[ \text{sorted}(v@pre, v\#1, l@pre, r@pre) \]

Assume:
\[ l@pre \geq r@pre \]

Then:
\[ \text{sorted}(v@pre, v@pre, l@pre, r@pre) \]
Our Approach to Design Verification (UML)

1. Specification and design
2. Automatic generation of questions
3. Design inspection (answering the questions)
4. Follow-up
Specification

Annotated use-case diagrams

- Actors, use-cases, relationships
- Use-case pre- and postconditions

requires
- The sender and the receiver are connected through the system.
- The sender has a valid message to transmit;

ensures
- The message has been transfered from the sender to the receiver;
Design

Annotated sequence diagrams

- Objects, messages, loops
- Pre- and postconditions, intermediate assertions, loop invariants

Annotated class diagrams

- Classes, associations
- Class invariants, operation pre- and postconditions

\[
\text{segment}(m) \quad \text{ensures} \quad \text{segmented}(\text{this}, \text{this}@\text{pre}, m, n)
\]
Question Generation

Use-case precondition

Use-case

Use-case postcondition

Sequence preconditions

Sequence diagrams

Sequence postconditions
Software Artifact Inspection Process

- Process:
  1. Planning phase
  2. Inspection phase (individual, voting, group meeting when no consens)
  3. Rework phase

- Advantages:
  - Systematic, documentable, repeatable, supports re-inspections
  - Asynchronous, distributed
Conclusions and Ongoing Work

• Works in principle (prototypes)
• But practical?
  ○ Complexity/amount of questions for real artifacts
  ○ Writing assertions for real artifacts