Run-Time Assertion Checking and Monitoring Java Programs

Envisage Bertinoro Summer School June 2014

June 19, 2014
Your Lecturers Today

Frank en Stijn
What This Talk Is All About

Formal Methods in Practice:

Theorie ist, wenn man alles weiß und nichts klappt.
Praxis ist, wenn alles klappt und keiner weiß warum.
What Fails In Practice: Run-Time Assertion Checking

Take for example **JML** (citing Peter Wong)

- Stability of tooling
- IDE support e.g. on-the-fly parsing and type checking, navigability between specifications and source codes
- Maintainability of specification due to constant code change
- Error reporting and analysis

See also

*Run-time checking of data- and protocol-oriented properties of Java programs: an industrial case study.*

*Stijn de Gouw, Frank S. de Boer, Peter Y. H. Wong and Einar Broch Johnsen. SAC 2013.*
Outline

Formal Specification: Assertions

Behavioral Abstraction

Run-time checking of data- and protocol-oriented properties

Tooling
Industrial Relevance

National Institute of Standards and Technology (NIST):

Software errors cost us approximately $60 billion per year in lost productivity, increased time to market costs, higher market transaction costs, etc. If allowed to continue unchecked this problem’s costs may get much worse.

Managerial Misconceptions:

Software development is not an art, and programmers are not artists, despite any claims to the contrary.

Management has come to believe the first and most important misconception: that it is impossible to ship software devoid of errors in a cost-effective way.
What Makes Software Buggy?

An *imperative* program describes *how* a problem can be solved by a computer.
The Von Neumann Architecture of Imperative Programming
What does the following program compute, assuming that the initial value of \( x \) is greater than or equal to 0?

\[
y := 0; \quad u := 0; \quad v := 1;
\]

\[
\textbf{while} \quad u + v \leq x
\]

\[
\textbf{do} \quad y := y + 1;
\]

\[
\quad u := u + v;
\]

\[
\quad v := v + 2
\]

\[
\textbf{od}
\]
Debugging: Let it Flow

What's the relation between the values of $x$, $y$, $u$ and $v$?
Robert Floyd Introduced Assertions For Program Specification in the Seventies

\[ y^2 \leq x < (y + 1)^2 \]
Edsger Dijkstra Introduced Structured Programming

*Debugging only shows that a program is incorrect.*
Sir. Tony Hoare Developed a First Programming Logic

\{P\}S\{Q\}
Design by Contract

Caller = Client and Callee = Supplier in Method calls in object-oriented programs

Designer must formally specify for each method:

- What does it expect? (precondition)
- What does it guarantee? (postcondition)
- What does it maintain? (invariant)

Main idea:

*Formal specification of contracts by assertions, i.e. logical formulas*
Design by Contract in Practice

- Object-oriented programming language **Eiffel** introduced by the company **Eiffel Software**.
- The Java Modelling Language **JML** supports **run-time assertion checking**.
- **Spec#** is a formal language for API contracts developed and used by Microsoft.
- **Object Constraint Language (OCL)** for the specification of **UML diagrams**.
Behavioral Abstraction (Information Hiding)

State of the Art (= state-based)

- Getters:
  \[\text{Get}_X\]

- Model variables (JML):
  \[\text{public model instance JMLObjectBag elementsInQueue}\]
Formal Semantics: Full Abstraction

Minimal information required for compositionality

That is, smallest congruence containing operational equivalence:

\[ S \equiv S' \text{ if and only if } O(C[S]) = O(C[S']), \]

for every context \( C[\cdot] \)
Compositionality Java Programs

Two perspectives:
- **Threads** (stack: shared-variable concurrency)
- **Classes (Objects)** (monitor: message passing)
Compositionality Shared Variable Concurrency (Multi-threading)

Initial/final state semantics is not compositional:

\[ O(x := x + 1; x := x + 1) = O(x := x + 2) \]

but

\[ O(x := x + 1; x := x + 1 \parallel x := 0) \neq O(x := x + 2 \parallel x := 0) \]

We need reactive sequences:

\[ R(x := x + 1) = \{ \langle \sigma, \sigma[x := \sigma(x) + 1] \rangle | \sigma \in \Sigma \} \]

and

\[ R(S_1 \parallel S_2) = R(S_1) \parallel R(S_2) \]

where \( \parallel \) denotes interleaving.

See

Compositional Proof Theory for Communicating Sequential Processes (CSP)

From non-compositional:

Communication Assumptions \( \{p\} c?x\{q\} \) and \( \{p\} c!e\{q\} \)

Cooperation Test

\[
\begin{align*}
\{p\} & \quad c!e \\
\{p'\} & \quad c?x
\end{align*}
\]

\[
\Rightarrow \{p \land p'\} x := e\{q \land q'\}
\]

to compositional by means of histories (or traces)

Communication Axioms

\[
\{\forall x. p[h \cdot (c, x)/h]\} c?x\{q\} \quad \text{and} \quad \{p[h \cdot (c, e)/h]\} c!e\{q\}
\]

Example:

\[
\begin{align*}
\{[h]_c = \epsilon\} c?x\{[h]_c = (c, x)\} & \quad \{[h]_c = \epsilon\} c!0\{[h]_c = (c, 0)\} \\
\{[h]_c = \epsilon\} c?x \parallel c!0\{[h]_c = (c, x) \land [h]_c = (c, 0)\}
\end{align*}
\]

See

*An assertion-based proof system for multithreaded Java*
by Abraham, de Boer, de Roever and Steffen, in TCS, Vol. 331, 2005.
A Short History of Histories

- Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes. Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, Martin Steffen: ICTAC 2004: 37-51
The Very Nature of Object-Orientation

Inherently *Parallel* (even *Sequential*)
Run-Time Assertion Checking

Requires

- Executable assertions

But what we want (need badly) is

*combining data- and protocol-oriented properties*
Grammars to specify protocols (= formal languages)

Main problem/challenge:

Integration grammars in assertion checking

that works in practice
Specifying Interfaces in Java: A Running Example

```java
interface Stack {
    void push(Object item);
    Object pop();
}
```
The Modelling Framework: Messages

call-push
  Attributes
    public Object item

return-push
  Attributes
    public Object item

call-pop
  Attributes
    public Object item

return-pop
  Attributes
    public Object result
Partial mappings from call and return events to tokens
Communication Views: An Example

```
view StackHistory {
    return void push(Object item) push,
    return Object pop() pop
}
```
General Properties of Communication Views

- Multiple views for interfaces
- Multiple views for classes/components (provided/required methods)
- User-defined event names
- Abstraction of irrelevant events
- Identifying different events
- Distinguishing different events using method signatures (method overloading)
class EList extends List {
    public EList append(Object element)
    public EList append(EList list) }

Elist stack

| S ::= push S₁ stack = S₁.stack.append(push.item) |
| S₁ S₂ stack = S₂.stack.append(S₁.Stack) |
| B stack = new EList() |
| B ::= push B pop |
| ϵ |
Example

Parse tree of sequence of tokens

push(5) push(7) pop(7)
interface Stack {
//@ public model instance StackHistory history;

//@ requires history.stack().size != 0;
//@ ensures history.stack() == \old(history.stack()).tail();
//@ ensures result == \old(history.stack()).head();
Object pop()
}
The Modelling Framework: Summary
Run-Time Assertion Checking: Method Invocation

Diagram:

- Program
- JML API
- History (instance)
- Parser

Connections:
- Program to JML API: Check Precondition
- JML API to History (instance): Get Attributes, Attribute values, Incoming Method Call
- History (instance) to Parser: Triggers, New Attribute values
- Program to JML API: Incoming Method Call
Run-Time Assertion Checking: Method Return
Attribute Grammars as Behavioral Types

Another Example: Specifying the **BufferedReader** class

```java
Class BufferedReader {
    BufferedReader(Reader in);
    void close();
    String readLine();
    ...
}
```

Communication View:

```java
view BufferedReaderHistory {
    new(Reader in) open,
    call String readLine() read,
    call void close() close
}
```
Extended Attribute Grammar modeling the behavior of a BufferedReader

- BufferedReader can only be read when opened and before closed.
- BufferedReader can only be closed by the object that opened it:

\[
\begin{align*}
S & ::= \text{open } C & \text{assert open.caller \neq null} \\
& \quad \implies \text{open.caller == C.caller;} \\
& | \quad \epsilon \\
C & ::= \text{read } C_1 & \text{C.caller = C}_1\text{.caller;} \\
& | \quad \text{close } S & \text{C.caller = close.caller;} \\
& | \quad \epsilon & \text{C.caller = null;}
\end{align*}
\]
Summarizing

Attribute grammars provide a systematic approach for specifying histories which

- allows a declarative expression of complex properties of histories and

- seamlessly combines specification of
  - protocol oriented properties (grammar)
  - data-oriented properties (attributes)

into a single formalism.