An Assertional Proof System for Multi-Threaded Java

Erika Ábrahám  Frank S. de Boer  Willem Paul de Roever  Martin Steffen

Christian-Albrechts University Kiel

Summer Research Institute, EPFL, July 2004
Overview

- Programming language $\text{Java}_{MT}$
- Assertion language
- Proof system
- Conclusion
Motivation

- safety-critical application areas
  → need for verification
- model checking: mostly for finite state systems
- existing deductive methods: mostly for sequential Java
Multithreading core of Java

Object of study: $\text{Java}_{MT}$

- heap-allocated objects, aliasing
- object creation
- method invocation, recursion, self-calls
- multithreading
- `wait` & `notify` monitor synchronization
- exceptions
- not covered (yet): inheritance, polymorphism . . .
Multithreading

- **threads** = sequential sequence of actions
- method calls/returns: **stack** of method bodies, each with **local** variables
- running in **parallel**
- **sharing** instance states
- **dynamically created** as instances of thread classes (+ explicitly **started**)
Monitors

- each object can act as monitor:
  - mutual exclusion between synchronized methods of a single instance
  - monitor coordination via methods: wait, notify, notifyAll
Abstract syntax

\[
\begin{align*}
\text{exp} & ::= x \mid u \mid \text{this} \mid \text{nil} \mid f(\text{exp}, \ldots, \text{exp}) \\
\text{stm} & ::= x := \text{exp} \mid u := \text{exp} \mid u := \text{new}^c \\
& \quad | \quad \text{exp}.m(\text{exp}, \ldots, \text{exp}); \text{receive} u \mid \text{exp}.\text{start}() \\
& \quad | \quad \epsilon \mid \text{stm}; \text{stm} \mid \text{if} \ \text{exp} \ \text{then} \ \text{stm} \ \text{else} \ \text{stm} \ \text{fi} \ldots \\
\text{modify} & ::= \text{nsync} \mid \text{sync} \\
\text{meth} & ::= \text{modify}m(u, \ldots, u)\{ \text{stm}; \text{return} \ \text{exp}\} \\
\text{meth}_{\text{predef}} & ::= \text{meth}_{\text{run}} \mid \text{meth}_{\text{start}} \mid \text{meth}_{\text{wait}} \mid \text{meth}_{\text{notify}} \mid \text{meth}_{\text{notifyAll}} \\
\text{class} & ::= c\{\text{meth} \ldots \text{meth} \mid \text{meth}_{\text{predef}}\} \\
\text{prog} & ::= \langle\text{class} \ldots \text{class} \mid \text{class}_{\text{main}}\rangle
\end{align*}
\]
### Semantics

- **straightforward structural operational semantics**
- **transitions between global configurations**

#### States

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>$\tau$</td>
<td>values of local variables</td>
</tr>
<tr>
<td>Global</td>
<td>$\sigma$</td>
<td>values of instance variables for each existing object</td>
</tr>
</tbody>
</table>

#### Configurations

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>$(\tau, stm)$</td>
<td>local state + point of exec.</td>
</tr>
<tr>
<td>Thread</td>
<td>$(\tau_0, stm_0) \ldots (\tau_n, stm_n)$</td>
<td>stack of local configurations</td>
</tr>
<tr>
<td>Global</td>
<td>$\langle T, \sigma \rangle$</td>
<td>set of thread configurations + global state</td>
</tr>
</tbody>
</table>
Impressionistic view on the SOS

\[
\beta = [e]^{(\alpha, \tau)}_{\xi} \in \text{dom}^2(\sigma) \quad \text{started}(T \cup \{\xi \circ (\alpha, \tau, e, \text{start}; stm)\}, \sigma) \quad \text{CALL}_{\text{start}}
\]

\[
\beta = [e]^{(\alpha, \tau)}_{\xi} \in \text{dom}^2(\sigma) \quad \text{started}(T \cup \{\xi \circ (\alpha, \tau, e, \text{start}; stm)\}, \sigma) \quad \text{CALL}_{\text{skip}}
\]

\[
\beta = [e]^{(\alpha, \tau)}_{\xi} \in \text{dom}^2(\sigma) \quad \text{started}(T \cup \{\xi \circ (\alpha, \tau, e, \text{start}; stm)\}, \sigma) \quad \text{CALL}_{\text{monitor}}
\]

\[
\beta = [e]^{(\alpha, \tau)}_{\xi} \in \text{dom}^2(\sigma) \quad \text{started}(T \cup \{\xi \circ (\alpha, \tau, e, \text{start}; stm)\}, \sigma) \quad \text{CALL}_{\text{wait}}
\]

\[
\beta = [e]^{(\alpha, \tau)}_{\xi} \in \text{dom}^2(\sigma) \quad \text{started}(T \cup \{\xi \circ (\alpha, \tau, e, \text{start}; stm)\}, \sigma) \quad \text{CALL}_{\text{signal}}
\]

\[
\beta = [e]^{(\alpha, \tau)}_{\xi} \in \text{dom}^2(\sigma) \quad \text{started}(T \cup \{\xi \circ (\alpha, \tau, e, \text{start}; stm)\}, \sigma) \quad \text{CALL}_{\text{signalAll}}
\]

E. Ábrahám, W.-P. de Roever, F. S. de Boer, M. Steffen  Java proof system
Semantics, e.g., instantiation

- instantiating a new object: \[ u := \text{new}^c \]
  - create a fresh object id (i.e., \( \beta \notin \text{dom}(\sigma) \))
  - initialize the instance state
  - extend the heap
  - store the new identity

\[
\begin{align*}
\beta &\fresh &\sigma_{\text{inst}} &= \sigma_{\text{inst}}^{c,\text{init}}[\text{this} \mapsto \beta] &\sigma' &= \sigma[\beta \mapsto \sigma_{\text{inst}}] \\
\langle T \cup \{ \xi \circ (\alpha, \tau, u := \text{new}^c; \text{stm}) \}, \sigma \rangle &\longrightarrow \langle T \cup \{ \xi \circ (\alpha, \tau[u \mapsto \beta], \text{stm}) \}, \sigma' \rangle
\end{align*}
\]

one thread
Proof-theoretical challenges

- dynamic object creation
- concurrency, multithreading
  - intra-object: shared variables concurrency
  - inter-object: communication via method calls, (self-calls)
  - monitor synchronization
The assertional proof system

- **proof outline**
  - augmentation by *auxiliary variables*/*bracketed sections*
  - *assertions*:
    - local *assertions* to all control points
    - class invariant for each class
    - global invariant

- **verification conditions** for
  - *initial correctness*
  - inductive step:
    - local correctness
    - interference freedom test
    - cooperation test
The assertion language

**local** sublanguage: properties of method execution

\[
\begin{align*}
\text{exp}_l & ::= z \mid x \mid u \mid \text{this} \mid \text{nil} \mid f(\text{exp}_l, \ldots, \text{exp}_l) \\
\text{ass}_l & ::= \text{exp}_l \mid \neg \text{ass}_l \mid \text{ass}_l \land \text{ass}_l \\
& \mid \exists z:\text{Int}. \text{ass}_l \ldots \\
& \mid \exists (z:\text{Object}) \in \text{exp}_l. \text{ass}_l \mid \exists (z:\text{Object}) \sqsubseteq \text{exp}_l. \text{ass}_l
\end{align*}
\]

**global** sublanguage: properties of communication/object structure

\[
\begin{align*}
\text{exp}_g & ::= z \mid \text{exp}_g.x \mid \text{nil} \mid f(\text{exp}_g, \ldots, \text{exp}_g) \\
\text{ass}_g & ::= \text{exp}_g \mid \neg \text{ass}_g \mid \text{ass}_g \land \text{ass}_g \mid \exists z. \text{ass}_g
\end{align*}
\]
Local correctness

**local** inductiveness for the executing local configuration (no communication):

\[
\begin{align*}
\models \mathcal{L} & \quad \text{pre}(\bar{y} := \bar{e}) \rightarrow \text{post}(\bar{y} := \bar{e})[\bar{e}/\bar{y}] \\
\models \mathcal{L} & \quad p \rightarrow l_c
\end{align*}
\]

for all assignments (outside bracketed sections) in class \(c\) with class invariant \(l_c\)
Interference freedom

- variables shared within one instance ⇒ interference
- when exactly can different “executions” interfere?
  - different threads, except matching signalling communication pairs
  - reentrant code pieces of the same thread, except matching return-communication

\[ \models \mathcal{L} \quad \text{pre}(\bar{y} := \bar{e}) \land q' \land \text{interferes}(q', \bar{y} := \bar{e}) \rightarrow q'[\bar{e}/\bar{y}] \]

where interferes(p, \bar{y} := \bar{e}) is defined as

\[
\begin{align*}
\text{thread} &= \text{thread}' \rightarrow \text{waits}_{\text{for}\_\text{ret}}(p, \bar{y} := \bar{e}) \land \\
\text{thread} \neq \text{thread}' &\rightarrow \neg \text{self}_{\text{start}}(p, \bar{y} := \bar{e}) .
\end{align*}
\]
Coop. test for communication (call)

... \{p_1\} \langle e_0.m(this, conf, thread, e); \{p_2\} \tilde{y}_1 := \tilde{e}_1; \{p_3\} \\
\langle \text{receive } u_{ret}; \{p_4\} \tilde{y}_4 := \tilde{e}_4; \{p_5\} ... \\

\{l_c\} \text{ sync } m \ (\text{caller, caller-thread, } u) \ {\{q_2\}} \\
\langle \text{conf := counter, counter := counter } + 1, \\text{thread := caller-thread,} \\
\text{lock := inc(lock), } \tilde{y}_2 := \tilde{e}_2; \{q_3\} \\
... \{q_4\} \\
\langle \text{return } e_{ret}; \{q_5\} \text{ lock := dec(lock), } \tilde{y}_3 := \tilde{e}_3 \rangle \} \{l_c\}
Coop. test for communication (call)

\[ \models g \quad GI \land P_1(z) \land Q'_1(z') \land \\
\text{comm} \land z \neq \text{nil} \land z' \neq \text{nil} \rightarrow \\
(P_2(z) \land Q'_2(z')) \circ f_{\text{comm}} \land \\
(GI \land P_3(z) \land Q'_3(z')) \circ f_{\text{obs2}} \circ f_{\text{obs1}} \circ f_{\text{comm}} \]

- $z, z'$: distinct fresh logical variables
- $\text{comm} =$

\[ (E_0(z) = z') \land (z'.\text{lock} = \text{free} \lor \text{thread}(z'.\text{lock}) = \text{thread}) \]

- $f_{\text{comm}} = [\tilde{E}(z), \text{Init}(\tilde{v})/\tilde{u}', \tilde{v}'],$ $f_{\text{obs1}} = [\tilde{E}_1(z)/z.\tilde{y}_1],$ $f_{\text{obs2}} = [\tilde{E}_2(z')/z'.\tilde{y}_2'].$
Coop. test for communication

- other kinds of communications: variations of the `comm`-assertion (and the “observations”):
  - `return`: must match caller and callee
  - `monitor`: callers must own the lock
  - `start` can be called (effectively) only once
  - return from a `wait`-method must re-acquire the lock
  - return from a `start`-method . . .
Coop. test for object creation

\[ \{p_1\} \langle u := \text{new}^c; \{p_2\} \bar{y} := \bar{e}\rangle \{p_3\} \]

- new object's id must be fresh
- heap extended \(\Rightarrow\) range of (unbounded) quantification changes

\[ \models_g \quad z \neq \text{nil} \land \]

\[ \exists z' : \text{list Object.} \left( \text{Fresh}(z', u) \land (Gl \land \exists u. P_1(z)) \downarrow z' \right) \rightarrow \]

\[ P_2(z) \land l_c(u) \land (Gl \land P_3(z)) \circ f_{\text{obs}} , \]

- \(\text{Fresh}(z', u) = \text{InitState}(u) \land u \not\in z' \land \forall v. v \in z' \lor v = u\)
Coop. test for notification

```java
{l_c} nsync wait (caller, caller_thread) { {q2}
    <conf := counter, counter := counter + 1, thread := caller_thread,
    wait := wait U {lock}, lock := free, \( \bar{y}'_2 := \bar{e}'_2 \)>;
    {q3}!signal {q4} \( \bar{y}' := \bar{e}' \); {q5}
    return getlock; {q6} lock := get(notified, thread),
    notified := notified \ get (notified, thread), \( \bar{y}'_3 := \bar{e}'_3 \)\{l_c}\n
{l_c} nsync notify (caller, caller_thread) { {p2}
    <conf := counter, counter := counter + 1, thread := caller_thread; \( \bar{y}'_2 := \bar{e}'_2 \)>;
    {p3}!signal {p4} notified := notified U get(wait, partner),
    wait := wait \ get (wait, partner), \( \bar{y} := \bar{e} \); {p5}
    return; {p6} \( \bar{y}'_3 := \bar{e}'_3 \)\{l_c}\n```
Coop. test for notification

\[ \models \quad p_3 \land q_3' \rightarrow (p_4 \land q_4') \circ f_{\text{comm}} \land (p_5 \land q_5') \circ f_{\text{obs}} \circ f_{\text{comm}}, \]

where \( f_{\text{comm}} = [{\text{thread'}}/\text{partner}] \),

- formulated in the local assertion language
- similar conditions for
  - \text{notifyAll} = \text{broadcast}
  - signalling \text{without} receiver

E. Ábrahám, W.-P. de Roever, F. S. de Boer, M. Steffen
Java proof system
Auxiliary variables

- thread/object identification: aux. formal parameters
  - caller’s object id
  - id of caller’s local configuration = “return address”\(^1\)
  - id of caller thread

- capture monitor discipline: aux. instance variables
  - lock : Object × Int + free
  - wait, notified : 2^{Object} × Int

⇒ The proof system is sound and (relative) complete

\(^1\) plus a mechanism to uniquely identify local configurations within an object, e.g., counter.
Related work

- Pierik, de Boer [4]
  - inheritance, subtyping
  - sequential
- de Boer, Amerika (Pool) [1] ...
- Poetzsch-Heffter, Müller [5], sequential Java.
- M. Huismann, B. Jacobs, et.al (Loop, PVS+Isabelle) [2] ...
- etc.
Conclusion

future/ongoing work:
  • inheritance, exceptions, etc
  • refined semantics: deadlock-sensitive
  • compositionality
  • PVS implementation
References I

Reasoning about dynamically evolving process structures. 

Reasoning about classes in object-oriented languages: Logical models and tools. 


A syntax-directed Hoare logic for object-oriented programming concepts. 
In Najm et al. [3], pages 64–78. 

A programming logic for sequential Java. 