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 $Java_{MT}$
- Assertion language
- Proof system
- Conclusion

Motivation

- safety-critical application areas
  $\longrightarrow$ need for verification
- model checking: mostly for finite state systems
- existing deductive methods: mostly for sequential *Java*

Multithreading core of *Java*

Object of study: Java_{MT}

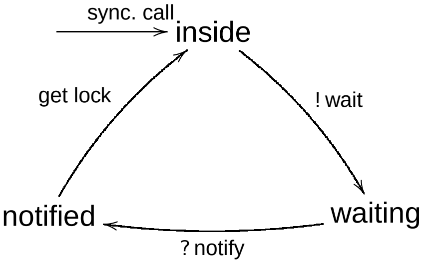
- heap-allocated objects, aliasing
- object creation
- method invocation, recursion, self-calls
- multithreading
- wait & notify monitor synchronization
- exceptions
- not covered (yet): inheritance, polymorphism . . .

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 (+ explicitely 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

$$
\begin{array}{rcl}
\mathit{exp} & ::= & x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(\mathit{exp}, \ldots, \mathit{exp}) \\
\mathit{stm} & ::= & x := \mathit{exp} \mid u := \mathit{exp} \mid u := \mathsf{new}^{c} \\
 & \mid & \mathit{exp}.m(\mathit{exp}, \ldots, \mathit{exp});\ \mathsf{receive}\ u \mid \mathit{exp}.\mathsf{start}() \\
 & \mid & \epsilon \mid \mathit{stm};\mathit{stm} \mid \mathsf{if}\ \mathit{exp}\ \mathsf{then}\ \mathit{stm}\ \mathsf{else}\ \mathit{stm}\ \mathsf{fi} \ldots \\
\mathit{modif} & ::= & \mathsf{nsync} \mid \mathsf{sync} \\
\mathit{meth} & ::= & \mathit{modif}\ m(u, \ldots, u)\{\ \mathit{stm};\ \mathsf{return}\ \mathit{exp}\} \\
\mathit{meth}_{\mathit{predef}} & ::= & \mathit{meth}_{\mathsf{run}}\ \mathit{meth}_{\mathsf{start}}\ \mathit{meth}_{\mathsf{wait}}\ \mathit{meth}_{\mathsf{notify}}\ \mathit{meth}_{\mathsf{notifyAll}} \\
\mathit{class} & ::= & c\{\mathit{meth} \ldots \mathit{meth}\ \mathit{meth}_{\mathit{predef}}\} \\
\mathit{prog} & ::= & \langle \mathit{class} \ldots \mathit{class}\ \mathit{class}_{\mathsf{main}} \rangle
\end{array}
$$

Semantics

- straightforward structural operational semantics
- transitions between global configurations

states

local	τ	values of local variables
global	σ	values of instance variables for each *existing* object

configurations

local	(τ, stm)	local state + point of exec.
thread	$(\tau_0, \mathit{stm}_0) \ldots (\tau_n, \mathit{stm}_n)$	stack of local configurations
global	$\langle T, \sigma \rangle$	set of thread configurations + global state

Impressionistic view on the SOS

$$\frac{\beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in \mathit{dom}^c(\sigma) \qquad \neg \mathit{started}(T \cup \{\xi \circ (\alpha, \tau, e.\mathrm{start}(); \mathit{stm})\}, \beta)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e.\mathrm{start}(); \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathit{stm}), (\beta, \tau_{\mathit{init}}^{\mathrm{start},c}, \mathit{body}_{\mathrm{start},c})\}, \sigma\rangle} \ \textsc{Call}_{\mathit{start}}$$

$$\frac{\beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in \mathit{dom}^c(\sigma) \qquad \mathit{started}(T \cup \{\xi \circ (\alpha, \tau, e.\mathrm{start}(); \mathit{stm})\}, \beta)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e.\mathrm{start}(); \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathit{stm})\}, \sigma\rangle} \ \textsc{Call}_{\mathit{start}}^{\mathit{skip}}$$

$$\frac{}{\langle T \,\dot\cup\, \{(\alpha, \tau, \mathrm{return})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{(\alpha, \tau, \epsilon)\}, \sigma\rangle} \ \textsc{Return}_{\mathit{start}}$$

$$\frac{\begin{array}{c} m \in \{\mathrm{wait}, \mathrm{notify}, \mathrm{notifyAll}\} \\ \beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in \mathit{dom}^c(\sigma) \qquad \mathit{owns}(\xi \circ (\alpha, \tau, e.m(); \mathit{stm}), \beta) \end{array}}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e.m(); \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathit{stm}) \circ (\beta, \tau_{\mathit{init}}^{m,c}, \mathit{body}_{m,c})\}, \sigma\rangle} \ \textsc{Call}_{\mathit{monitor}}$$

$$\frac{\neg \mathit{owns}(T, \beta)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathrm{receive}; \mathit{stm}) \circ (\beta, \tau', \mathrm{return}_{\mathit{getlock}})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathit{stm})\}, \sigma\rangle} \ \textsc{Return}_{\mathit{wait}}$$

$$\frac{}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, !\mathrm{signal}; \mathit{stm})\} \,\dot\cup\, \{\xi' \circ (\alpha, \tau', ?\mathrm{signal}; \mathit{stm}')\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathit{stm})\} \,\dot\cup\, \{\xi' \circ (\alpha, \tau', \mathit{stm}')\}, \sigma\rangle \end{array}} \ \textsc{Signal}$$

$$\frac{\mathit{wait}(T, \alpha) = \emptyset}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, !\mathrm{signal}; \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathit{stm})\}, \sigma\rangle} \ \textsc{Signal}_{\mathit{skip}}$$

$$\frac{T' = \mathit{signal}(T, \alpha)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, !\mathrm{signal\_all}; \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T' \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathit{stm})\}, \sigma\rangle} \ \textsc{SignalAll}$$

Semantics, e.g., instantiation

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

$$\frac{\beta \text{ } fresh \qquad \sigma_{inst} = \sigma_{inst}^{c,init}[\text{this} \mapsto \beta] \qquad \sigma' = \sigma[\beta \mapsto \sigma_{inst}]}{\langle T \,\dot{\cup}\, \underbrace{\{\xi \circ (\alpha, \tau, u{:=}\text{new}^c; stm)\}}_{\text{one thread}}, \sigma\rangle \longrightarrow \langle T \,\dot{\cup}\, \{\xi \circ (\alpha, \tau[u \mapsto \beta], stm)\}, \sigma'\rangle} \text{ NEW}$$

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

$$
\begin{array}{lcl}
exp_l & ::= & z \mid x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp_l, \ldots, exp_l) \\
ass_l & ::= & exp_l \mid \neg ass_l \mid ass_l \wedge ass_l \\
 & \mid & \exists z{:}\mathsf{Int}.\, ass_l \ldots \\
 & \mid & \exists (z{:}\mathsf{Object}) \in exp_l.\, ass_l \mid \exists (z{:}\mathsf{Object}) \sqsubseteq exp_l.\, ass_l
\end{array}
$$

global sublanguage: properties of communication/object structure

$$
\begin{array}{lcl}
exp_g & ::= & z \mid exp_g.x \mid \mathsf{nil} \mid \mathsf{f}(exp_g, \ldots, exp_g) \\
ass_g & ::= & exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists z.\, ass_g
\end{array}
$$

Local correctness

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

$$\begin{aligned} &\models_{\mathcal{L}} \quad pre(\vec{y} := \vec{e}) \rightarrow post(\vec{y} := \vec{e})[\vec{e}/\vec{y}] \\ &\models_{\mathcal{L}} \quad p \rightarrow I_c \end{aligned}$$

for all assignments (outside bracketed sections) in class c with class invariant I_c

Interference freedom

- variables shared within one instance $\Rightarrow$ 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

$$\models_{\mathcal{L}} \quad pre(\vec{y} := \vec{e}) \wedge q' \wedge \text{interferes}(q', \vec{y} := \vec{e}) \rightarrow q'[\vec{e}/\vec{y}]$$

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

$$\begin{array}{l} \text{thread} = \text{thread}' \rightarrow \text{waits\_for\_ret}(p, \vec{y} := \vec{e}) \wedge \\ \text{thread} \neq \text{thread}' \rightarrow \neg\text{self\_start}(p, \vec{y} := \vec{e})\,. \end{array} \quad s$$

Coop. test for communication (call)

$\ldots$ $\{p_1\}$ $\langle$ `e0.m(this,conf,thread,e);` $\{p_2\}$ $\vec{y}_1 := \vec{e}_1\rangle;$ $\{p_3\}$
$\langle$ `receive` $u_{ret};$ $\{p_4\}$ $\vec{y}_4 := \vec{e}_4\rangle;$ $\{p_5\}$ $\ldots$

$\{I_c\}$ `sync m (caller,caller_thread,u) {` $\{q_2\}$
$\langle$ `conf := counter, counter := counter + 1,`
`thread := caller_thread,`
`lock := inc(lock),` $\vec{y}_2 := \vec{e}_2\rangle;$ $\{q_3\}$
$\ldots$ $\{q_4\}$
$\langle$ `return` $e_{ret};$ $\{q_5\}$ `lock := dec(lock),` $\vec{y}_3 := \vec{e}_3\rangle$ `}` $\{I_c\}$

Coop. test for communication (call)

$$
\begin{array}{ll}
\models_{\mathcal{G}} & GI \wedge P_1(z) \wedge Q'_1(z') \wedge \\
& \text{comm} \wedge z \neq \text{nil} \wedge z' \neq \text{nil} \rightarrow \\
& \quad (P_2(z) \wedge Q'_2(z')) \circ f_{comm} \wedge \\
& \quad (GI \wedge P_3(z) \wedge Q'_3(z')) \circ f_{obs2} \circ f_{obs1} \circ f_{comm}
\end{array}
$$

- z, z': distinct fresh logical variables
- comm =

$$(E_0(z) = z') \wedge (z'.\text{lock} = \text{free} \vee \text{thread}(z'.\text{lock}) = \text{thread})$$

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

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

$$\{p_1\}\langle u := \mathrm{new}^c; \{p_2\}\vec{y} := \vec{e}\rangle\{p_3\}$$

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

$$\begin{aligned}\models_{\mathcal{G}} \quad & z \neq \mathrm{nil} \wedge \\ & \exists z' : \mathrm{list\ Object}. \Big(\mathrm{Fresh}(z', u) \wedge (GI \wedge \exists u.\, P_1(z)) \downarrow z'\Big) \rightarrow \\ & \qquad P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z)) \circ f_{obs} \ ,\end{aligned}$$

- $\mathrm{Fresh}(z', u) = \mathrm{InitState}(u) \wedge u \notin z' \wedge \forall v.\, v \in z' \vee v = u$

Coop. test for notification

$\{I_c\}$ `nsync wait (caller,caller_thread) {` $\{q_2\}$
$\langle$conf $:=$ counter, counter $:=$ counter $+ 1$, thread $:=$ caller_thread,
`wait := wait` $\cup$ `{lock}, lock := free,` $\vec{y}_2' := \vec{e}_2'\rangle$;
$\{q_3\}\langle$?signal; $\{q_4\}$ $\vec{y}' := \vec{e}'\rangle$; $\{q_5\}$
$\langle$ `return`$_{getlock}$ `;` $\{q_6\}$ lock $:=$ get(notified, thread),
`notified := notified \ get(notified,thread),` $\vec{y}_3' := \vec{e}_3'\rangle\{I_c\}$

$\{I_c\}$ `nsync notify (caller,caller_thread) {` $\{p_2\}$
$\langle$conf $:=$ counter, counter $:=$ counter $+ 1$, thread $:=$ caller_thread; $\vec{y}_2 := \vec{e}_2\rangle$;
$\{p_3\}\langle$!signal $\{p_4\}$ `notified := notified` $\cup$ `get(wait,partner),`
`wait := wait \ get(wait,partner),` $\vec{y} := \vec{e}\rangle$; $\{p_5\}$
$\langle$ `return;` $\{p_6\}$ $\vec{y}_3 := \vec{e}_3\rangle\{I_c\}$

Coop. test for notification

$$\models_{\mathcal{L}} \quad p_3 \wedge q_3' \quad \rightarrow \quad (p_4 \wedge q_4') \circ f_{comm} \wedge (p_5 \wedge q_5') \circ f_{obs} \circ f_{comm} \ ,$$

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

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

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 : $\text{Object} \times \text{Int} + \text{free}$
  - wait, notified : $2^{\text{Object} \times \text{Int}}$

$\Rightarrow$ *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

[1] P. America and F. S. de Boer.
Reasoning about dynamically evolving process structures.
Formal Aspects of Computing, 6(3):269–316, 1993.

[2] U. Hensel, M. Huisman, B. Jacobs, and H. Tews.
Reasoning about classes in object-oriented languages: Logical models and tools.
In C. Hankin, editor, *Proceedings of ESOP '98*, volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[3] E. Najm, U. Nestmann, and P. Stevens, editors.
Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '03), Paris, volume 2884 of *Lecture Notes in Computer Science*. Springer-Verlag, Nov. 2003.

[4] C. Pierik and F. S. de Boer.
A syntax-directed Hoare logic for object-oriented programming concepts.
In Najm et al. [3], pages 64–78.
An extended version appeared as University of Utrecht Technical Report UU-CS-2003-010.

[5] A. Poetzsch-Heffter and P. Müller.
A programming logic for sequential Java.
In S. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.