Verification for Java's Monitor Concept

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

Centrum voor Wiskunde en Informatica
Amsterdam

Inst. für Informatik u. Prakt. Mathematik
Christian-Albrechts Universtität zu Kiel

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: $Java_{MT}$

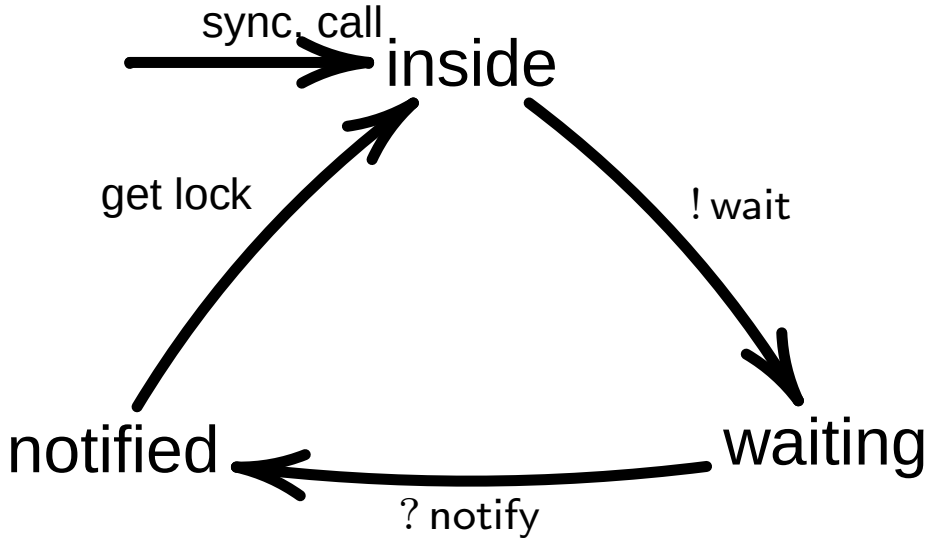
- heap-allocated objects, aliasing
- object creation
- method invocation, recursion, self-calls
- multithreading
- wait & notify monitor synchronization
- not covered (yet): inheritance, polymorphism, exceptions . . .

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

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

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	$(\tau_0, stm_0) \ldots (\tau_n, stm_n)$	stack of local configurations
global	$\langle T, \sigma \rangle$	set of thread configurations + global state

Impressionistic view on SOS

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

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

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

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

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

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

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

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

Semantics, e.g., instantiation

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

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

Proof-theoretical challenges

- dynamic object creation
- concurrency, multithreading
  - shared variables concurrency within objects
  - 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

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

global sublanguage: properties of communication

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

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

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

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

$$(\mathsf{thread} = \mathsf{thread}' \rightarrow \mathsf{waits\_for\_ret}(p, \vec{y} := \vec{e})) \wedge$$
$$(\mathsf{thread} \neq \mathsf{thread}' \rightarrow \neg\mathsf{gets\_signalled}(p, \vec{y} := \vec{e})) \ .$$

Coop. test for communication (call)

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

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

Coop. test for communication (call)

$$\begin{aligned}\models_{\mathcal{G}} \quad & GI \wedge P_1(z) \wedge Q_1'(z') \wedge \\ & \text{communicating} \wedge z \neq \mathsf{nil} \wedge z' \neq \mathsf{nil} \rightarrow \\ & (P_2(z) \wedge Q_2'(z')) \circ f_{comm} \wedge \\ & (GI \wedge P_3(z) \wedge Q_3'(z')) \circ f_{obs2} \circ f_{obs1} \circ f_{comm}\end{aligned}$$

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

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

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

Coop. test for communication

- other kinds of communications: variations of the communicating-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

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

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

$$\models_{\mathcal{G}} \quad z \neq \mathsf{nil} \wedge$$

$$\exists z' : \mathsf{list}\,\mathsf{Object}.\left(\mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\, P_1(z)) \downarrow z'\right) \rightarrow$$

$$P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z)) \circ f_{obs}\ ,$$

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

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; $\vec{y}_2 :=$

$\{p_3\}\langle$!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 $f_{comm} = [\{\mathsf{thread}'\}/\mathsf{partner}]$,

- 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" [a]
  - id of caller thread
- capture monitor discipline: aux. instance variables
  - lock : $\mathsf{Object} \times \mathsf{Int} + \mathsf{free}$
  - wait, notified : $2^{\mathsf{Object} \times \mathsf{Int}}$

$\Rightarrow$ The proof system is sound and (relative) complete

[a] and mechanism for unique identifi er of local confi gurations within an object, e.g., counter.

Related work

- de Boer, Amerika (Pool) [AdB93] . . .
- Poetzsch-Heffter, Müller [PHM99], sequential *Java*.
- M. Huismann, B. Jacobs, et.al (Loop, PVS+Isabelle) [HHJT98] . . .
- etc.

Conclusion

Future work:

- inheritance etc
- refined semantics: deadlocks-sensitive
- component-based extension
- compositionality
- PVS implementation under way

References

[AdB93] Pierre America and Frank S. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1993.

[ÁMdBdRS02a] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Compositional operational semantics of Java$_{MT}$. Technical Report TR-ST-02-2, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2002.

[ÁMdBdRS02b] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verifi cation for Java's monitor concept: Soundness and completeness. Technical Report TR-ST-02-3, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 2002. to appear.

[ÁMdBdRS02c] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen.

Verifi cation for Java's reentrant multithreading concept. In Mogens Nielsen and Uffe H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, April 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

[HHJT98] J. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Chris Hankin, editor, *Proceedings of ESOP '98*, volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.