Verification for Java’s Monitor Concept

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

Centrum voor Wiskunde en Informatica  Christian-Albrechts-Universität
Amsterdam, The Netherlands  Kiel, Germany

Albert-Ludwigs-Universität
Freiburg, Germany
Motivation

- Safety-critical application areas → need for verification
- Model checking: mostly for finite state systems
- Existing deductive methods: mostly for sequential Java
Example

```java
account {
    int balance;
    ...
    sync get_money(amount) {
        bool ok;
        ok := (amount<=balance);
        if ok then balance := balance - amount;
        return ok;
    }
}
```

Invariant property: each instance of the class has a non-negative balance.
Overview

- Programming language $Java_{MT}$
- Assertion language
- Proof system
- Conclusion
Multithreading core of Java

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

- Dynamic object creation, aliasing
- Method invocation, recursion, self-calls
- **Multithreading**
  - Threads are *dynamically created*
  - Running in *parallel*
  - **Sharing** instance states
  - Call chain: *stack* of method executions with their *local* states
- **Synchronization, monitor** concept (methods *wait*, *notify*, and *notifyAll*)
- Not covered (yet): inheritance, polymorphism, exceptions . . .
• Each object can act as monitor:
  • **Mutual exclusion** between *synchronized* methods of a single instance (per thread, reentrant)
  • Monitor **coordination** via methods: `wait`, `notify`, `notifyAll`
Abstract syntax

\[ e ::= x | u | \text{this} | \text{nil} | f(e, \ldots, e) \]

\[ \text{stm} ::= x := e | u := e | u := \text{new}^c \]

\[ e.m(e, \ldots, e); \text{receive} u | e.\text{start}() \]

\[ e | \text{stm; stm} | \text{if} e \text{ then stm else stm fi . . .} \]

\[ \text{modify} ::= \text{nsync} | \text{sync} \]

\[ \text{meth} ::= \text{modify}m(u, \ldots, u)\{ \text{stm; return} e \} \]

\[ \text{meth}_\text{run} ::= \text{modify}run()\{ \text{stm; return} \} \]

\[ \text{meth}_\text{predef} ::= \text{meth}_\text{start} \text{meth}_\text{wait} \text{meth}_\text{notify} \text{meth}_\text{notifyAll} \]

\[ \text{class} ::= c\{ \text{meth} \ldots \text{meth} \text{meth}_\text{run} \text{meth}_\text{predef} \} \]

\[ \text{prog} ::= \langle \text{class} \ldots \text{class} \text{class}_\text{main} \rangle \]
Example

... e0.m(e); receive u; ...

sync m(u') {
    body;
    return e';
}

Java Monitors – p.8
States and configurations

\[ \sigma : \]

\[
\begin{array}{c}
\sigma(\alpha) \\
\sigma(\beta) \\
\ldots
\end{array}
\]

instance state

\[ T : \] stack of a thread

\[
\begin{array}{c}
(\alpha_n, \tau_n, stm_n) \\
\ldots \\
(\alpha_0, \tau_0, stm_0)
\end{array}
\]

local configuration

\[
\begin{array}{c}
(\alpha'_m, \tau'_m, stm'_m) \\
\ldots \\
(\alpha'_0, \tau'_0, stm'_0)
\end{array}
\]

global state
Semantics: Method call

- Invokes \( m(\tilde{u})\{\text{body}\} \) of \( \beta = [e_0]^{\sigma(\alpha), \tau} \neq \text{nil} \)
- \( m \notin \{\text{start, wait, notify, notifyAll}\} \) synchronized → lock of \( \beta \) is free or the caller owns it
- \( m \in \{\text{wait, notify, notifyAll}\} \) → the caller owns \( \beta \)'s lock
- \( m = \text{start} \) → new stack, no receive
Proof-theoretical challenges

- Dynamic **object creation**
- **Concurrency**, multithreading
  - **Intra**-object: Shared variables concurrency
  - **Inter**-object: method calls (self-calls)
  - Monitor synchronization
The assertional proof system

- Proof outline
  - Augmentation by auxiliary variables
  - Annotation with assertions
- Verification conditions for
  - Initial correctness
  - Inductive step:
    - Local correctness
    - Interference freedom test
    - Cooperation test
The assertion language

Local sublanguage: properties of method execution
describes instance+local states

\[ e ::= z \mid x \mid u \mid \text{this} \mid \text{nil} \mid f(e, \ldots, e) \]
\[ p ::= e \mid \neg p \mid p \land p \mid \exists z: \text{Int}. p \]
\[ \mid \exists (z: \text{Object}) \in e. p \mid \exists (z: \text{Object}) \sqsubseteq e. p \]

Global sublanguage: properties of communication
describes global states

\[ E ::= z \mid E.x \mid \text{nil} \mid f(E, \ldots, E) \]
\[ P ::= E \mid \neg P \mid P \land P \mid \exists z. P \]
Augmentation

- **Rules of semantics**: in terms of global configurations
- **Assertional logic**: in terms of states
- **Completeness**: requires expressibility of the semantics in the logic
- **Solution**:  
  - augmentation with fresh auxiliary variables without influencing the original control flow  
- **Simultaneous observations**: bracketed sections  
  \[ \langle \text{stm}; \bar{y} := \bar{e} \rangle \]
... ⟨ e0.m(e); \vec{y}_1 := \vec{e}_1⟩; ⟨ receive u; \vec{y}_4 := \vec{e}_4⟩; ...

sync m(u’) {
  ⟨\vec{y}_2 := \vec{e}_2⟩;
  body;
  ⟨ return e’; \vec{y}_3 := \vec{e}_3⟩
}
### Specific auxiliary variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Kind</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>conf</td>
<td>local</td>
<td>local configuration identity (object unique)</td>
</tr>
<tr>
<td>caller</td>
<td>formal par.</td>
<td>“return address” = caller object + caller’s conf</td>
</tr>
<tr>
<td>thread</td>
<td>formal par.</td>
<td>thread identity, thread of $\alpha$ gets the identity $\alpha$</td>
</tr>
<tr>
<td>lock</td>
<td>instance</td>
<td>lock’s owner + nr synchr. calls</td>
</tr>
<tr>
<td>wait</td>
<td>instance</td>
<td>waiting threads + nr synchr. calls</td>
</tr>
<tr>
<td>notified</td>
<td>instance</td>
<td>notified threads + nr synchr. calls</td>
</tr>
<tr>
<td>started</td>
<td>instance</td>
<td>thread already started?</td>
</tr>
</tbody>
</table>

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

- a local assertion to each control point
- a local assertion $I$ called class invariant to each class
- a global assertion $GI$ called global invariant to the program
Example

\[ ... \{p_1\} \langle e_0.m((\text{this}, \text{conf}), \text{thread}, e)\rangle; \{p_3\} \]
\[ \langle \text{receive } u_{ret}\rangle; \{p_5\} ... \]

\{I_c\} \quad \text{sync } m (\text{caller}, \text{thread}, u) \{ \{p_2\} \]
\[ \langle \text{conf := counter, counter := counter + 1, } \]
\[ \text{lock := inc(lock)}\rangle; \{p_3\} \]
\[ \text{body}; \quad \{p_4\} \]
\[ \langle \text{return } e_{ret}; \{p_5\} \quad \text{lock := dec(lock)}\rangle \} \{I_c\} \]
Example

\{I_c\} \text{nsync wait (caller,thread) \{ } \{p_2\} \\
\text{\{conf := counter, counter := counter + 1,} \\
\text{ wait := wait \cup lock, lock := free\}; } \{p_3\} \\
\text{\{return\}_getlock;} \{p_4\} \text{ lock := get(notified,thread),} \\
\text{ notified := notified \setminus get(notified,thread)\}\}\{I_c\}

\{I_c\} \text{nsync notify (caller,thread) \{ } \{p_2\} \\
\text{\{conf := counter, counter := counter + 1\}; } \{p_3\} \\
\text{ wait, notified := notify(wait,notified); } \{p_4\} \\
\text{\{return\} \} } \{I_c\}
The proof system

LC : local correctness
IFT : interference freedom test
COOP : cooperation test

<table>
<thead>
<tr>
<th>Local conf.</th>
<th>Assignment</th>
<th>Communication</th>
<th>Object creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executing</td>
<td>LC (local)</td>
<td>COOP (global)</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td>IFT (local)</td>
<td></td>
</tr>
</tbody>
</table>
Local correctness

For all assignments $\bar{y} := \bar{e}$ outside bracketed sections

$$\models_\mathcal{L} \{\text{pre}(\bar{y} := \bar{e})\} \quad \bar{y} := \bar{e} \quad \{\text{post}(\bar{y} := \bar{e})\}$$

For all assertions $p$ in a class $c$

$$\models_\mathcal{L} p \rightarrow I_c$$
**COOP: Method call**

<table>
<thead>
<tr>
<th>$\sigma(\alpha)$</th>
<th>$\sigma(\beta)$</th>
<th>$\ldots$</th>
<th>$\sigma'(\alpha)$</th>
<th>$\sigma'(\beta)$</th>
<th>$\ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle \alpha, \tau, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; \text{stm} \rangle$</td>
<td>$\ldots$</td>
<td>$\rightarrow$</td>
<td>$\langle \beta, \tau'', \text{body} \rangle$</td>
<td>$\langle \alpha, \tau', \text{stm} \rangle$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>

- $\beta = \left[ e_0 \right]_{\vec{e}}^{(\alpha),\tau} \neq \text{nil}$ has method $m(\vec{u}) \{ \langle \vec{y}_2 := \vec{e}_2 \rangle; \text{body} \}$
- Method call is **enabled**
- Computation step consists of
  - **initializing** the callee’s local state,
  - **communicating** the parameters,
  - executing $\vec{y}_1 := \vec{e}_1$ by the caller, and
  - executing $\vec{y}_2 := \vec{e}_2$ by the callee.
Express in the logic: $\beta = \llbracket e_0 \rrbracket^{(\alpha), \tau}_{e} \neq \text{nil}$ and method call enabled

$z$ is caller: $\{p_1\}$ $\langle e_0.m(\bar{e}); \{p_2\} \bar{y}_1 := \bar{e}_1 \rangle \{p_3\}$

$z'$ is callee: $\{q_1 = I\}$ $m(\bar{u})\{ q_2 \} \langle \bar{y}_2 := \bar{e}_2 \rangle; \{q_3\}$ body

communicating: $E_0(z) = z' \wedge z' \neq \text{nil} \wedge$ synch matching communication pair enabled communication

<table>
<thead>
<tr>
<th>synch</th>
<th>for</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>$m \notin {\text{start, wait, notify, notifyAll}}$ non-synchr.</td>
</tr>
<tr>
<td>$z'.\text{lock} = \text{free} \displaystyle \vee \text{owns(thread, z'.lock)}$</td>
<td>$m \notin {\text{start, wait, notify, notifyAll}}$ synchr.</td>
</tr>
<tr>
<td>$\text{owns(thread, z'.lock)}$</td>
<td>$m \in {\text{wait, notify, notifyAll}}$</td>
</tr>
</tbody>
</table>
**COOP: Method call**

\[ z \text{ is caller: } \{ p_1 \} \quad \langle e_0.m(\vec{e}); \{ p_2 \} \quad \vec{y}_1 := \vec{e}_1 \rangle \quad \{ p_3 \} \]

\[ z' \text{ is callee: } \{ q_1 = I \} \quad m(\vec{u})\{ \{ q_2 \} \quad \langle \vec{y}_2 := \vec{e}_2; \{ q_3 \} \quad \text{body} \} \]

\[
|\models \{ GI \wedge P_1(z) \wedge Q'_1(z') \wedge \text{communicating} \}
\]

\[
\vec{u}', \vec{v}' := \vec{E}(z), \text{Init}(\vec{v})
\]

**init + communication**

\[
\{ P_2(z) \wedge Q'_2(z') \}
\]
**COOP: Method call**

\( z \) is caller: \( \{ p_1 \} \langle e_0.m(\overline{e})\rangle \{ p_2 \} \overline{y}_1 := \overline{e}_1 \) \{ p_3 \}

\( z' \) is callee: \( \{ q_1 = I \} \) \( m(\overline{u})\) \( \{ q_2 \} \langle \overline{y}_2 := \overline{e}_2 \rangle ; \) \( \{ q_3 \} \) body

\[ \models_{G} \{ GI \land P_1(z) \land Q'_1(z') \land \text{communicating} \} \]

\( \overline{u}', \overline{v}' := \overline{E}(z), \text{Init}(\overline{v}) \)

init + communication

\( z.\overline{y}_1 := \overline{E}_1(z) \)

caller observation

\( z'.\overline{y}_2 := \overline{E}'_2(z') \)

callee observation

\( \{ GI \land P_3(z) \land Q'_3(z') \} \)
Example: COOP

(1) \{own\(\text{thread, lock}\)\}
(2) \langle \text{this.wait((this,conf),thread)}; 
(3) \{\neg\text{own}\(\text{thread, lock}\)\} \ldots 

(4) \{true\} \text{nsync wait (caller,thread) } \{ 
(5) \{\text{own}\(\text{thread, lock}\)\}
(6) \langle \ldots \text{lock := free} \ldots \rangle; 
(7) \{\neg\text{own}\(\text{thread, lock}\)\} \ldots \} 

\models_{g} \{\text{own}\(\text{thread, z.lock}\) \land z' = z\} \quad (1) \text{ + communicating parameter passing} 
thread' := \text{thread}, \ldots 
\{\text{own}\(\text{thread', z'.lock}\)\} \quad (5)
Example: COOP

(1) \( \{ \text{owns(thread, lock)} \} \)

(2) \( \langle \text{this.wait((this,conf),thread)}; \)

(3) \( \{ \neg \text{owns(thread, lock)} \} \ldots \)

(4) \( \{ \text{true} \} \) \text{nsync wait (caller,thread) } \{ \)

(5) \( \{ \text{owns(thread, lock)} \} \)

(6) \( \langle \ldots \text{lock := free } \ldots ; \rangle \)

(7) \( \{ \neg \text{owns(thread, lock)} \} \ldots \} \)

\( \models_g \{ \text{owns(thread, z.lock)} \land z' = z \} \)

\text{thread' := thread, } \ldots ; z'.lock := \text{free, } \ldots \)

\( \{ \neg \text{owns(thread, z.lock)} \land \neg \text{owns(thread', z'.lock)} \} \)

(1) + communicating

\text{comm + callee obs}

(3) + (7)
Interference freedom test

- **LC+COOP**: invariance for the *executing* configurations
- **IFT**: invariance of assertions under execution of others
- Interference freedom means invariance
  - of assertions $p$ outside bracketed sections
  - under assignments $y := e$ (also inside bracketed sections)
  - in the same object,
  - if $p$ is not active in the step executing $y := e$. 
Interference freedom test

For all \( p \) at a control point and \( \vec{y} := \vec{e} \) occurring in the same class,

\[
\models_{\mathcal{L}} \quad \{p' \land \text{pre}(\vec{y} := \vec{e}) \land \text{interleavable}(p, \vec{y} := \vec{e})\}
\]

\[
\vec{y} := \vec{e}
\]

\[
\{p'\},
\]

where \text{interleavable}(p, \vec{y} := \vec{e}) \) expresses that the local configuration of \( p \) is not active in the step executing \( \vec{y} := \vec{e} \), i.e.,

- it is not the one executing the assignment, and
- if the assignment observes communication, then it is neither the communication partner.
interleavable\( (p, \bar{y} := \bar{e}) \):

if (thread = thread') then

\[ receive(p) \land (return(\bar{y} := \bar{e}) \rightarrow (\text{caller} \neq (\text{this}, \text{conf}'))) \]

else

\[ \text{caller} \neq (\text{this}, \text{conf}') \]
Example: IFT

(1) \{owns(thread, lock)\}
(2) \langle\text{this.wait((this, conf), thread)}\rangle;
(3) \{\neg\text{owns(thread, lock)}\} \ldots

(4) \{true\} \text{nsync wait (caller, thread)}\{
(5) \{owns(thread, lock)\}
(6) \langle\ldots \text{lock} := \text{free} \ldots\rangle;
(7) \{\neg\text{owns(thread, lock)}\} \ldots\}

\models L \{owns(thread', lock) \land owns(thread, lock) \land \text{thread} \neq \text{thread}'\}
\ldots \text{lock} := \text{free} \ldots \{owns(thread', lock)\}
Future work:

- PVS implementation under way
- Synchronized statements
- Inheritance etc.
- Verifying deadlock freedom
- Component-based extension, compositionality
### Global vs. Local Annotation

#### Global View:

<table>
<thead>
<tr>
<th>Global view:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(α, τ₁, {P₁}y := e; {Q₁}stm₁)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>(β, τᵢ, {Pᵢ}stmᵢ)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>(α, τⱼ, {Pⱼ}stmⱼ)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

#### Local View:

<table>
<thead>
<tr>
<th>Local view:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(α, τ₁, {p₁}y := e; {q₁}stm₁)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>(α, τᵢ, {pᵢ}stmᵢ)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>(α, τⱼ, {pⱼ}stmⱼ)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Java Monitors – p.33