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, The Netherlands

Christian-Albrechts-Universtität
Kiel, Germany

Albert-Ludwigs-Universität
Freiburg, Germany

Motivation

- Safety-critical application areas
  $\rightarrow$ need for verification
- Model checking: mostly for finite state systems
- Existing deductive methods: mostly for sequential *Java*

Example

```
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

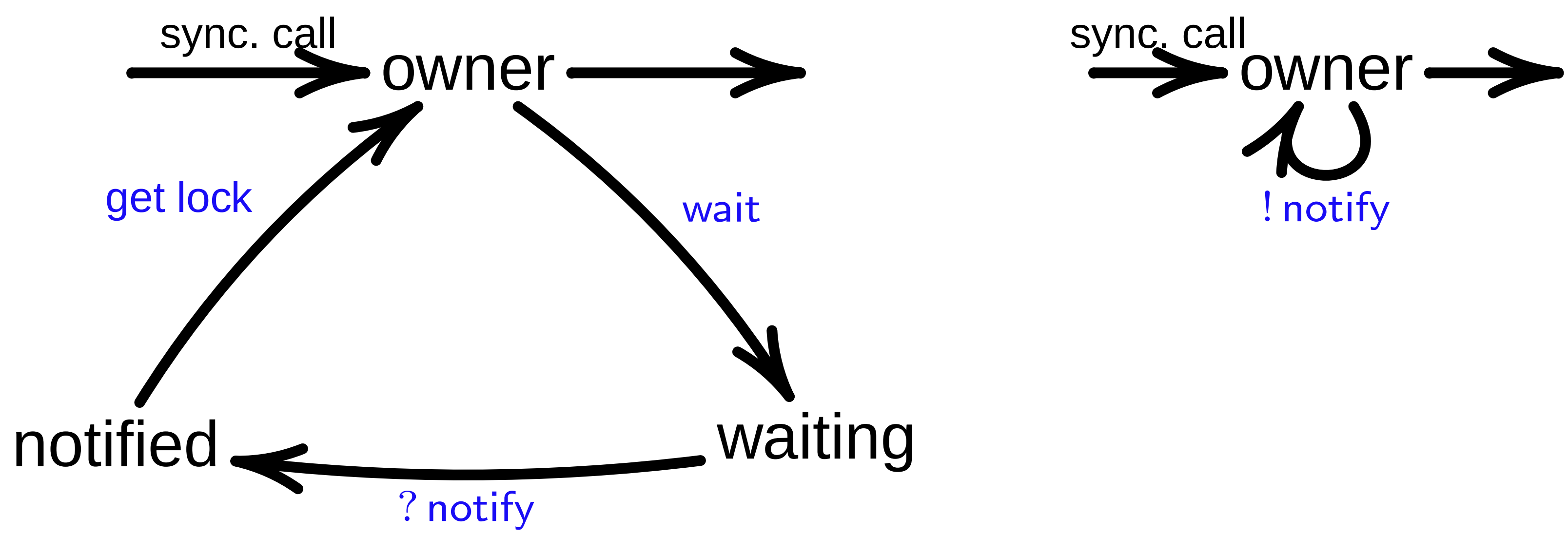
Multithreading core of Java

Object of study: *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 . . .

Monitors

- 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

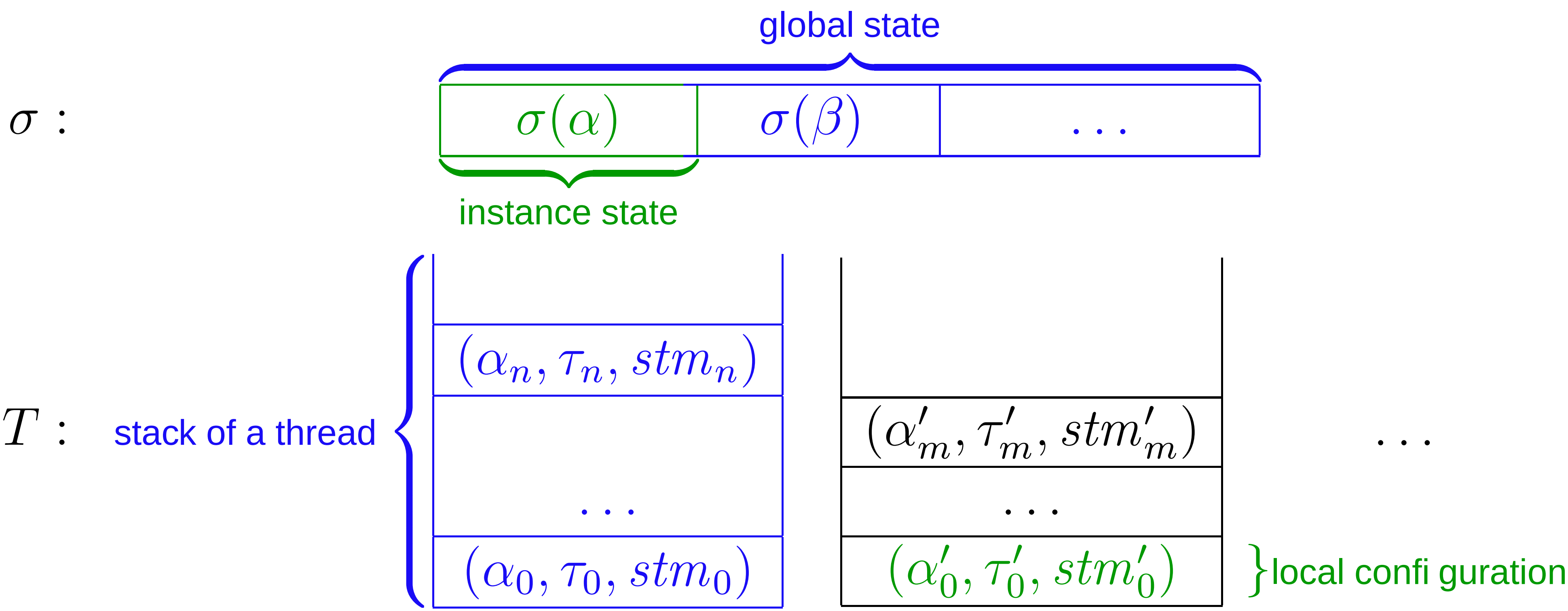
$$
\begin{array}{rcl}
e & ::= & x \mid u \mid \textsf{this} \mid \textsf{nil} \mid \textsf{f}(e, \ldots , e) \\
stm & ::= & x := e \mid u := e \mid u := \textsf{new}^c \\
 & \mid & e.m(e, \ldots, e); \textsf{receive}\, u \mid e.\textsf{start}() \\
 & \mid & \epsilon \mid stm; stm \mid \textsf{if}\ e\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

Example

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

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

States and configurations



Semantics: Method call

$\boxed{e_0.m(\vec{e})}$

- Invokes $m(\vec{u})\{body\}$ of $\beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \neq nil$
- $m \notin \{\text{start}, \text{wait}, \text{notify}, \text{notifyAll}\}$ synchronized $\longrightarrow$ lock of β is free or the caller owns it
- $m \in \{\text{wait}, \text{notify}, \text{notifyAll}\} \longrightarrow$ the caller owns β's lock
- $m = \text{start} \longrightarrow$ new stack, no receive

$\sigma(\alpha)$	$\sigma(\beta)$	$\ldots$

$(\alpha, \tau, e_0.m(\vec{e}); \text{receive } u_{ret}; stm)$
$\ldots$

$\ldots \qquad \longrightarrow$

$\sigma(\alpha)$	$\sigma(\beta)$	$\ldots$

$(\beta, \tau_{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}], body)$
$(\alpha, \tau, \text{receive } u_{ret}; stm)$
$\ldots$

$\ldots$

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

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

Global sublanguage: properties of communication
describes global states

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

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 $\langle stm; \vec{y} := \vec{e} \rangle$

Example

$\ldots$ $\langle$ **e0.m(e);** $\vec{y}_1 := \vec{e}_1\rangle$; $\langle$ **receive u;** $\vec{y}_4 := \vec{e}_4\rangle$; **...**

sync m(u’) {

  $\langle\vec{y}_2 := \vec{e}_2\rangle$**;**

  body;

  $\langle$ **return e’;** $\vec{y}_3 := \vec{e}_3\rangle$

}

Specific auxiliary variables

Name	Kind	Meaning
conf	local	local configuration identity (object unique)
caller	formal par.	“return address” = caller object + caller’s conf
thread	formal par.	thread identity, thread of α gets the identity α
lock	instance	lock’s owner + nr synchr. calls
wait	instance	waiting threads + nr synchr. calls
notified	instance	notified threads + nr synchr. calls
started	instance	thread already started?

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

Annotation

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

$\ldots \{p_1\}$ $\langle$ `e0.m((this,conf),thread,e)`$\rangle$; $\{p_3\}$

$\langle$ **receive** $u_{ret}\rangle$; $\{p_5\}$ ...

$\{I_c\}$ **sync m (caller,thread,u) {** $\{p_2\}$

$\langle$ `conf := counter, counter := counter + 1,`

`lock := inc(lock)`$\rangle$; $\{p_3\}$

body; $\{p_4\}$

$\langle$ **return** e_{ret}; $\{p_5\}$ `lock := dec(lock)`$\rangle$ **}** $\{I_c\}$

Example

$\{I_c\}$ **`nsync wait (caller,thread) {`** $\{p_2\}$

$\langle$ **`conf := counter, counter := counter + 1,`**

`wait := wait` $\cup$ **`lock, lock := free`**$\rangle$; $\{p_3\}$

$\langle$ **`return`**$_{getlock}$; $\{p_4\}$ **`lock := get(notified,thread),`**

`notified := notified` $\setminus$ **`get(notified,thread)`**$\rangle\}\{$

$\{I_c\}$ **`nsync notify (caller,thread) {`** $\{p_2\}$

$\langle$ **`conf := counter, counter := counter + 1`**$\rangle$; $\{p_3\}$

`wait, notified := notify(wait,notified)`; $\{p_4\}$

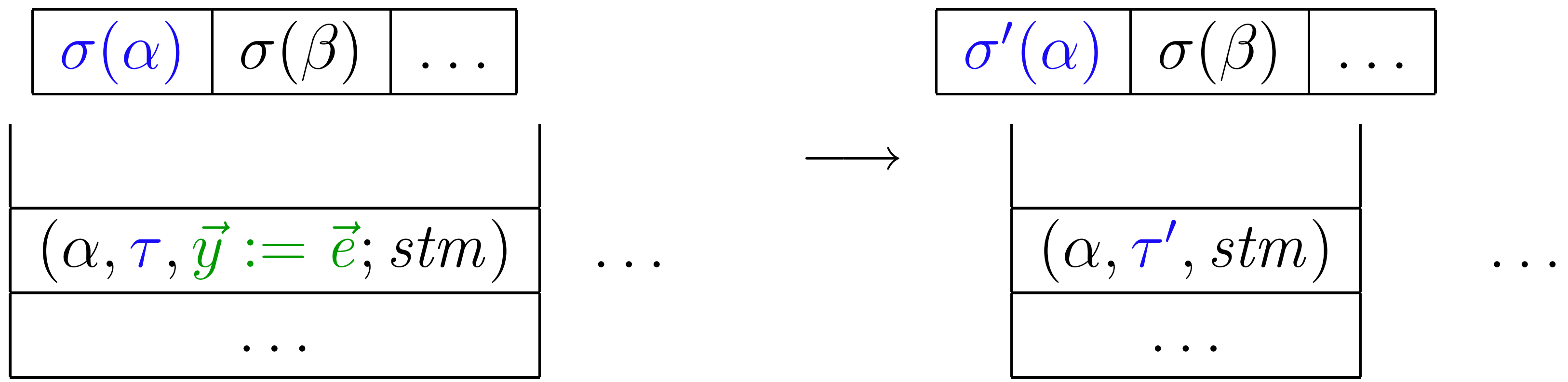
$\langle$ **`return`**$\rangle$ **`}`** $\{I_c\}$

The proof system

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

Local conf.	Assignment	Communication	Object creation
Executing	LC (local)	COOP (global)	
Other	IFT (local)		

Local correctness



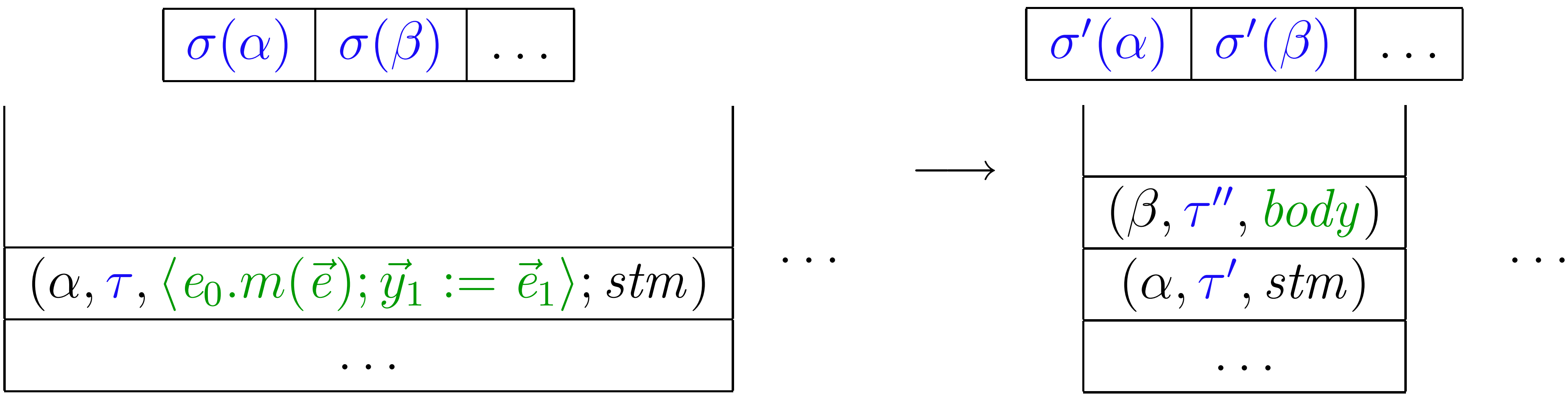
For all assignments $\vec{y} := \vec{e}$ outside bracketed sections

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

For all assertions p in a class c

$$\models_{\mathcal{L}} \quad p \rightarrow I_c$$

COOP: Method call



- $\beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \neq nil$ has method $m(\vec{u})\ \{\langle \vec{y}_2 := \vec{e}_2 \rangle; body\}$
- 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.

COOP: Method call

Express in the logic: $\beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \neq nil$ and method call enabled

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

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

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

synch :	for
true	$m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ non-synchr.
$z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{owns}(\mathsf{thread}, z'.\mathsf{lock})$	$m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ synchr.
$\mathsf{owns}(\mathsf{thread}, z'.\mathsf{lock})$	$m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$

COOP: Method call

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

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

$$\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{communicating}\}$$

$$\underbrace{\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})}_{\text{init + communication}}$$

$$\{P_2(z) \wedge Q_2'(z')\}$$

COOP: Method call

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

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

$$\models_{\mathcal{G}} \quad \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \textsf{communicating}\}$$

$$\underbrace{\vec{u}', \vec{v}' := \vec{E}(z), \textsf{Init}(\vec{v});}_{\text{init + communication}} \quad \underbrace{z.\vec{y}_1 := \vec{E}_1(z);}_{\text{caller observation}} \quad \underbrace{z'.\vec{y}_2' := \vec{E}_2'(z')}_{\text{callee observation}}$$

$$\{GI \wedge P_3(z) \wedge Q_3'(z')\}$$

Example: COOP

$$
\begin{array}{ll}
(1) & \{owns(thread, lock)\} \\
(2) & \langle\ \texttt{this.wait((this,conf),thread)}\rangle; \\
(3) & \{\neg owns(thread, lock)\} \quad \ldots \\
 & \\
(4) & \{true\} \quad \texttt{nsync wait (caller,thread) \{} \\
(5) & \{owns(thread, lock)\} \\
(6) & \langle \ldots \quad \texttt{lock := free} \quad \ldots \rangle; \\
(7) & \{\neg owns(thread, lock)\} \ldots \ \texttt{\}}
\end{array}
$$

$$
\begin{array}{ll}
\models_{\mathcal{G}} \{\mathsf{owns}(\mathsf{thread}, z.\mathsf{lock}) \wedge z' = z\} & (1) + \text{communicating} \\
\quad \mathsf{thread}' := \mathsf{thread}, \ldots & \textbf{parameter passing} \\
\quad \{\mathsf{owns}(\mathsf{thread}', z'.\mathsf{lock})\} & (5)
\end{array}
$$

Example: COOP

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

(2) $\langle$ `this.wait((this,conf),thread)`$\rangle;$

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

(4) $\{true\}$ **`nsync wait (caller,thread) {`**

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

(6) $\langle\ldots$ **`lock := free`** $\ldots\rangle;$

(7) $\{\neg owns(thread, lock)\} \ldots$ **`}`**

$$\models_{\mathcal{G}} \quad \{\mathsf{owns}(\mathsf{thread}, z.\mathsf{lock}) \wedge z' = z\}$$ $(1) +$ communicating

$$\mathsf{thread}' := \mathsf{thread}, \ldots ; z'.\mathsf{lock} := \mathsf{free}, \ldots$$ **comm** $+$ **callee obs**

$$\{\neg\mathsf{owns}(\mathsf{thread}, z.\mathsf{lock}) \wedge \neg\mathsf{owns}(\mathsf{thread}', z'.\mathsf{lock})\}$$ $(3) + (7)$

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 $\vec{y} := \vec{e}$ (also inside bracketed sections)
  - in the same object,
  - if p is not active in the step executing $\vec{y} := \vec{e}$.

Interference freedom test

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

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

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.

Interference freedom test

interleavable$(p, \vec{y} := \vec{e})$:

$$
\begin{array}{l}
\text{if } (\text{thread} = \text{thread}')\, \text{then} \\
\qquad \mathit{receive}(p) \wedge (\mathit{return}(\vec{y} := \vec{e}) \rightarrow (\text{caller} \neq (\text{this}, \text{conf}'))) \\
\text{else} \\
\qquad \text{caller} \neq (\text{this}, \text{conf}')
\end{array}
$$

Example: IFT

$$
\begin{array}{ll}
(1) & \{owns(thread, lock)\} \\
(2) & \langle \texttt{this.wait((this,conf),thread)} \rangle; \\
(3) & \{\neg owns(thread, lock)\} \quad \ldots \\
 & \\
(4) & \{true\} \ \texttt{nsync wait (caller,thread)}\{ \\
(5) & \{owns(thread, lock)\} \\
(6) & \langle \ldots \quad \texttt{lock} := \texttt{free} \quad \ldots \rangle; \\
(7) & \{\neg owns(thread, lock)\} \ldots \}
\end{array}
$$

$$
\models_{\mathcal{L}} \quad \{\mathsf{owns}(\mathsf{thread}', \mathsf{lock}) \land \mathsf{owns}(\mathsf{thread}, \mathsf{lock}) \land \mathsf{thread} \neq \mathsf{thread}'\}
$$

$$
\ldots \mathsf{lock} := \mathsf{free} \ldots
$$

$$
\{\mathsf{owns}(\mathsf{thread}', \mathsf{lock})\}
$$

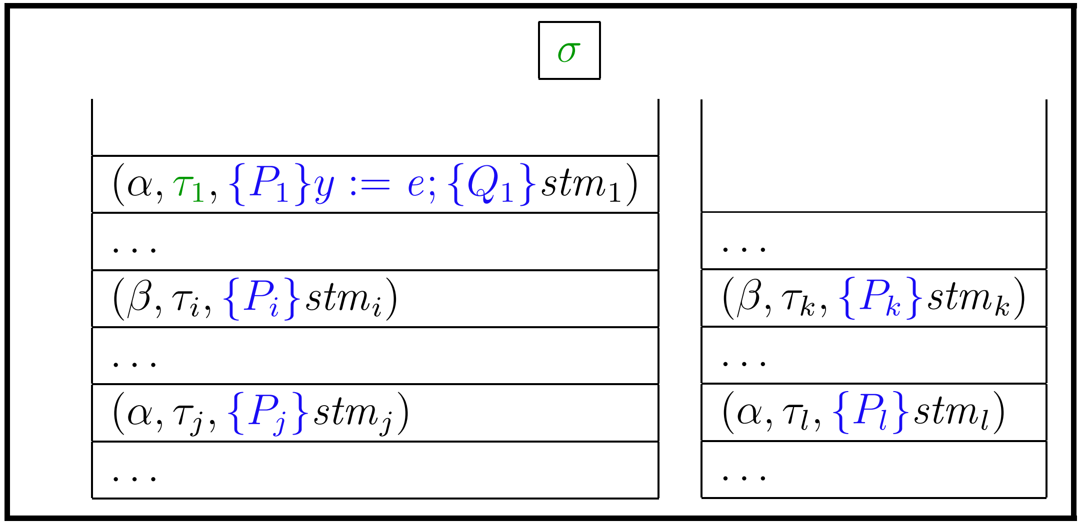
Conclusion

Future work:

- PVS implementation under way
- Synchronized statements
- Inheritance etc.
- Verifying deadlock freedom
- Component-based extension, compositionality

Global vs. local annotation

Global view:



Local view: