Abstract Interface Behavior of Object-Oriented Languages with Monitors

Martin Steffen

Christian-Albrechts University Kiel

Oslo

20 February 2006
Structure

introduction

semantics

interface description
  lock ownership
  data dependencies
  control dependencies

conclusion
introduction

semantics

interface description
lock ownership
data dependencies
control dependencies

conclusion
Introduction

- considered so far
  - classes and instantiation
    ⇒ heap
  - multithreading (vs. sequential/deterministic programs)
  - connectivity

- here: synchronization/monitors
Monitors

- shared (instance) state + concurrency $\Rightarrow$ mutex
- sync. mechanism: monitors
- for instance in Java
- here
  - no synchronized blocks
  - no wait/signal\(^1\)
  - no connectivity
- but:
  - re-entrant monitors (recursion)
- deliverable for task 1 ("compositionality and modularity: a semantic approach"), subtask 1.c ("basic features: libraries and synchronization protocols"), cf. [2, Sec. 7.2].

\(^1\)In Java: wait and notify.
Why is this interesting?

- fundamental question: 
  
  *what is observable of an oo program?*

- Now:

  *Does the addition of monitors increase or decrease the discriminating power or not?*

- intuitively: 2 plausible answers:
  - the observer sees less!
  - the observer sees more!
Why is this interesting?

- fundamental question: what is observable of an oo program?

- Now:
  
  Does the addition of monitors increase or decrease the discriminating power or not?

- intuitively: 2 plausible answers:
  - the observer sees less!
  - the observer sees more!
Why is this interesting?

- fundamental question: 
  \[\text{what is observable of an oo program?}\]

- Now:
  \[\text{Does the addition of} \ \text{monitors increase or decrease the discriminating power or not?}\]

- intuitively: 2 plausible answers:
  - the observer sees less!
  - the observer sees more!
Road map

- incorporate monitors into the semantics
- characterization of the interface behavior
  - may and must approximation of lock-ownership
- design goals
  - (preferably) seamless extension of the calculus with an eye to compositionality
  - clean separation of concerns between assumptions vs. commitments

- intuitively:
  - enabledness of input must depend only on the environment (= assumption)
  - enabledness of output must depend only on the component (= commitments)
- interface trace must contain all relevant information relevant (and not part of the internal state(s))
- cf. game theory
introduction

semantics

interface description
  lock ownership
  data dependencies
  control dependencies

conclusion
Syntax

- modest changes
- objects with locks
- extend object = class + fields (written $o[c, F]$ to “class + fields + lock”)

$$o[c, F, n]$$

(lock $n = \text{reference to thread}$)
Syntax

\begin{align*}
C & ::= \mathbf{0} \mid C \parallel C \mid \nu(n:T).C \mid n[0] \mid n[n, F, n] \mid n\langle t \rangle & \text{program} \\
O & ::= F, M & \text{object} \\
M & ::= l^u = m, \ldots, l^u = m, l^s = m, \ldots, l^s = m & \text{method suite} \\
F & ::= l^u = f, \ldots, l^u = f & \text{fields} \\
m & ::= \varsigma(n:T).\lambda(x:T, \ldots, x:T).t & \text{method} \\
f & ::= \varsigma(n:T).\lambda().v \mid \varsigma(n:T).\lambda().\bot & \text{field} \\
t & ::= v \mid \text{stop} \mid \text{let } x:T = e \text{ in } t & \text{thread} \\
e & ::= t \mid \text{if } v = v \text{ then } e \text{ else } e \mid \text{if } \text{undef}(v.l) \text{ then } e \text{ else } e & \text{expr.} \\
& \quad \mid v.l(v, \ldots, v) \mid v.l := v \mid \text{currentthread} \\
& \quad \mid \text{new } n \mid \text{new}\langle t \rangle \\
v & ::= x \mid n & \text{values}
\end{align*}
Semantics

1. operational semantics
2. remember the design-goals
3. two stages
   - internal semantics
     - closed system
     - spec. of the “virtual machine”
   - external semantics
     - interaction with environment via
     - message passing (calls/returns)
first attempt

• example: incoming call of unsynchronized method

\[ \Xi = \Xi + a \quad \Xi \vdash [a] : T \]

\[ a = \nu(\Xi'). \ n\langle \text{call } o_r.l(\vec{v})? \rangle \]

\[ t_{\text{blocked}} = \text{let } x':T' = \text{block in } t \]

\[ \Xi \vdash C \parallel n\langle t_{\text{blocked}} \rangle \xrightarrow{a} \]

\[ \Xi \vdash C \parallel C(\Theta') \parallel n\langle \text{let } x:T = o_r.l(\vec{v}) \text{ in return } x; t_{\text{blocked}} \rangle \]
first attempt

- example: incoming call of synchronized method
- assume: lock is free

\[ \Xi = \Xi + a \quad \Xi \vdash [a] : T \]

\[ a = \nu(\Xi'). \ n\langle \text{call o_r.l(\vec{v})} \rangle? \quad t_{\text{blocked}} = \text{let } x':T' = \text{block in } t \]

\[ \Xi \vdash C \parallel o[c, F', \bot_{\text{thread}}] \parallel n\langle t_{\text{blocked}} \rangle \xrightarrow{a} \]

\[ \Xi \vdash C \parallel C(\Theta') \parallel o[c, F', n] \parallel n\langle \text{let } x:T = o_r.l(\vec{v}) \text{ in return } x; \ t_{\text{blocked}} \rangle \]
first attempt

- example: incoming call of synchronized method
- assume: lock is free

\[
\dot{\Xi} = \Xi + a \quad \dot{\Xi} \vdash [a] : T
\]

\[
a = \nu(\Xi'). \; n\langle \text{call } o_r.l(\vec{v}) \rangle? \quad t_{\text{blocked}} = \text{let } x':T' = \text{block in } t
\]

\[
\Xi \vdash C \parallel o[c, F', \perp_{\text{thread}}] \parallel n\langle t_{\text{blocked}} \rangle \xrightarrow{a}
\]

\[
\dot{\Xi} \vdash C \parallel C(\Theta') \parallel o[c, F', n] \parallel n\langle \text{let } x:T = o_r.l(\vec{v}) \text{ in return } x; t_{\text{blocked}} \rangle
\]

- problem:
  - internal and external behavior not separated
  - whether the incoming call is possible: dependent on the component-internal state, i.e.,
  - the history trace doesn’t contain enough information to determine enabledness

\[\text{Note: for } t_{\text{blocked}}, \text{ the problem is not there even if it looks the same.}\]
“Non-atomic lock grabbing”

- handing over of call:
  - irrespective of availability of lock
  - i.e., no difference of external/interfaces rules for synchronized vs. non-synchronized methods!
  - component is input enabled

⇒ lock-grabbing (of comp. locks) is an internal step

- interface interaction: non-atomic lock-handling.
“Non-atomic lock grabbing”

- handing over of call:
  - irrespective of availability of lock
  - i.e., no difference of external/interfaces rules for synchronized vs. non-synchronized methods!
  - component is input enabled

⇒ lock-grabbing (of comp. locks) is an internal step

- interface interaction: non-atomic lock-handling.

\[ \hat{\Xi} = \Xi + a \quad \hat{\Xi} \vdash [a] : T \]

\[ a = \nu(\Xi'). \ n\langle \text{call o_r.l(\vec{v})} \rangle? \quad t_{\text{blocked}} = \text{let } x': T' = \text{block in } t \]

\[ \Xi \vdash C \parallel n\langle t_{\text{blocked}} \rangle \xrightarrow{a} \]

\[ \hat{\Xi} \vdash C \parallel C(\Theta') \parallel n\langle \text{let } x : T = \text{o_r.l(\vec{v}) in return x; } t_{\text{blocked}} \rangle \]
Internal steps

\[c[(F, M) \parallel o[c, F', \perp_{\text{thread}}] \parallel n\langle \text{let } x : T = o.I^s(\vec{v}) \text{ in } t \rangle \xrightarrow{\tau} \]

\[c[(F, M) \parallel o[c, F', n] \parallel n\langle \text{let } x : T = M.I^s(o)(\vec{v}) \text{ in } \text{release}(o); t \rangle \]

\[c[(F, M) \parallel o[c, F', n] \parallel n\langle \text{let } x : T = o.I^s(\vec{v}) \text{ in } t \rangle \xrightarrow{\tau} \]

\[c[(F, M) \parallel o[c, F', n] \parallel n\langle \text{let } x : T = M.I^s(o)(\vec{v}) \text{ in } t \rangle \quad \text{CALL}_{i_2}^s \]

- 2 internal rules for sync. methods
- note: re-entrancy, aux. syntax release
introduction

semantics

interface description
  lock ownership
  data dependencies
  control dependencies

conclusion
Interface description: Task

- cf. Andreas’ talk
- characterize possible interface behavior
- possible = adhering to the restriction of the language
  - well-typed
  - no violation of mutex
- rudimentary trace logic
Example (1)

- 2 calls, competing for the same (component) lock
- data dependence
  - $o'$ received by the first call (of $n_1$)
  - returned by second thread $n_1$ afterwards
- note: $o'$ is new

program environment

$\gamma_{c_1}$

$\gamma_{c_2}$

$\gamma_{c_1}'$

$\gamma_{r_2}$

- question: is that trace possible?
Example (1)

- 2 calls, competing for the same (component) lock
- data dependence
  - \(\nu o'\) received by the first call (of \(n_1\))
  - returned by second thread \(n_1\) afterwards
- note: \(o'\) is **new**

\[
\gamma_{c_1} \gamma_{c_2} \gamma'_{c_1} \gamma'_{c_2} = (\nu o' : c) n_1 \langle \text{call } o.l(o') \rangle? n_2 \langle \text{call } o.l() \rangle? n_1 \langle \text{call } \tilde{o}.l() \rangle! n_2 \langle \text{return}(o') \rangle!
\]

- question: *is that trace possible?*
Example (1)

\[ \gamma c_1 \land \gamma c_2 \land \gamma' c_1 \land \gamma' c_2 \land (\nu o') n_1 \langle \text{call } o.l(o') \rangle? n_2 \langle \text{call } o.l() \rangle? n_1 \langle \text{call } \tilde{o}.l() \rangle! n_2 \langle \text{return}(o') \rangle! \]

- question: *is that trace possible?*
- the answer is **no!**
- **data:** "\(n_1 \text{ before } n_2\)"
- **monitors:**
  - the outgoing call of \(n_1\) shows that \(n_1\) must have the lock now
    \[ \Rightarrow \ n_2 \text{ cannot have it now: } \Rightarrow \]
    "\(n_2 \text{ before } n_1\)"
Example (1)

\[ \gamma_{c_1} \gamma_{c_2} \gamma'_{c_1} \gamma_{r_2} = (\nu o':c)n_1\langle \text{call } o.l(o')\rangle? n_2\langle \text{call } o.l()\rangle? n_1\langle \text{call } o.o.l()\rangle! n_2\langle \text{return}(o')\rangle! \]

- question: is that trace possible?

\[ \gamma_{c_1} \gamma_{c_2} \]

(2)

Note: non-atomic lock-grabbing ⇒ no order!
Example (1)

\[
\gamma_{c_1} \gamma_{c_2} \gamma'_{c_1} \gamma_{r_2} = \\
(\nu o':c)n_1\langle\text{call o.l}(o')\rangle? n_2\langle\text{call o.l}()\rangle? n_1\langle\text{call ñ.l}()\rangle! n_2\langle\text{return}(o')\rangle!
\]

- question: is that trace possible?

\[
\gamma_{c_1} \gamma_{c_2} \\
\downarrow n_1 \\
\gamma'_{c_1}! \\
\]

(3)

Note: there is no order between events of \(n_1\) and \(n_2\)!
Example (1)

\[ \gamma_{c_1} \gamma_{c_2} \gamma'_{c_1} \gamma_r \]

\[ (\nu o' : c)n_1 \langle \text{call } o.l(o') \rangle? n_2 \langle \text{call } o.l() \rangle? n_1 \langle \text{call } \tilde{o}.l() \rangle! n_2 \langle \text{return}(o') \rangle! \]

- question: *is that trace possible?*

\[\neg\]

(4)

**Note:**
- data dependence because of *o'*
Conditions characterizing monitors

• apart from conditions concerning non-monitor features
  • well-typedness
  • freshness
  • (connectivity)

• 3 types of dependencies/precedences between events

1. **mutual exclusion:**
   If a thread has taken the lock of a monitor, interactions of other threads with that monitor must either occur before the lock is taken, or after it has been released again.

2. **data dependencies:**
   No value (unless generated new) can be transmitted before it has been received.

3. **control dependencies:**
   Within 1 thread, the events are linearly ordered.
Lock ownership

- question:
  
  Given interaction of thread $n$, is the lock of object $o$ available?

- first attempt:
  
  “After call $n$\langle call o.l()\rangle?$, thread $n$ owns the lock of $o$.”

- alas: not true!

- complication: non-atomic lock-grabbing
  
  Handing-over call $\Rightarrow$ not necessarily obtaining lock
Lock ownership: non-atomic lock grabbing

- delayed observation:
  
  \[ n\langle \text{call } o.l()\rangle? \]
  
  “after \[ n\langle \text{call } o.l()\rangle? \], thread \( n \) may own lock of component object \( o \).”

- and later:
  
  \[ n\langle \text{call } o.l()\rangle? n\langle \text{call } o'.l()\rangle! \]
  
  “after \[ n\langle \text{call } o.l()\rangle? n\langle \text{call } o'.l()\rangle! \], thread \( n \) must own lock of \( o \).

- 2 approximations per thread:
  
  - potential lock-ownership: “may”, written: \( \diamondsuit n o \)
  - necessary lock-ownership: “must”, written: \( \Box n o \)
Lock-ownership: May-approximation

- given the trace $t$ projected to one thread
- from the component-perspective\(^3\)

after $s$, the thread may own the lock of $o$: \(\triangledown t s : \lozenge o\)

\(^3\)dually for the environment.
Lock-ownership: May-approximation

\[ \vdash s_2 : balanced \quad s_2 \neq \epsilon \quad \triangleright s_1 : \diamond o \]

\[ \triangleright s_1 \ s_2 : \diamond o \quad \text{M-}\diamond \]

receiver(\(s_1 \gamma_c\)) = o

\[ \triangleright s_1 \ \gamma_c? : \diamond o \quad \text{M-I}\diamond_1 \]

receiver(\(s_1 \gamma_c\)) \neq o

\[ \triangleright s_1 \ \gamma_c? : \diamond o \quad \text{M-I}\diamond_2 \]

\[ \triangleright s_1 : \diamond o \quad \text{M-O}\diamond \]

\[ \triangleright s_1 \ \gamma_c! : \diamond o \]
Lock-ownership: Must-approximation

• similar system as in the may case
• based on the may-system\textsuperscript{3}
• again from the component-perspective
  after \( s \), the thread must own the lock of \( o \):

\[ \triangledown \vdash s : \Box o \]

\textsuperscript{3}but no mutual recursion
Lock-ownership: Must-approximation

\(
\begin{align*}
\text{M-I}\Box_1 & \quad \text{M-I}\Box_2 \\
\Gamma \vdash t : \Box o & \quad \Gamma \vdash t : \Box o \\
\Gamma \vdash t_{c?} : \Box o & \quad \Gamma \vdash t_{r?} : \Box o \\
\Gamma \vdash t_{c!} : \Diamond o & \quad \Gamma \vdash t_{r!} : \Box o \\
\text{M-O}\Box_1 & \quad \text{M-O}\Box_2
\end{align*}
\)
Illustration

Example

\[ t = \gamma_c ? = (\nu \Xi) n \langle \text{call o_r.l(o)} \rangle ? . \]

then

\[ \Xi \vdash t : \diamond o_r \quad \text{and} \quad \Xi \vdash t : \neg \diamond o \]

Note: \( \diamond \) is a \textit{local} interpretation.

Example

\[ t = \gamma_c ? \gamma_r ! = (\nu \Xi) n \langle \text{call o_r.l()} \rangle ? n \langle \text{return()} \rangle ! . \]

Then:

\[ \Xi \vdash \gamma_c ? : \diamond n o_r \quad \text{but} \quad \Xi \nvdash \gamma_c ? : \Box n o_r \]

and

\[ \Xi \vdash t : \neg \diamond o_r . \]
Mutual exclusion

- here: again for component locks
- “global” perspective: not just one thread
- mutex precedence edges for event $a$ after $r$ wrt. component object $o$.

$$M_{\Theta}(ra, o)$$

- auxiliary definitions:
  - “after may”: $\diamondsuit(t, o)$
  - “before must”: $\square(t, o)$
- edges: $\vdash a_1 \rightarrow^m a_2$
- distinction for $a$ between
  - incoming communication
    - no condition for incoming returns
    - incoming calls
  - outgoing communication: 2 conditions
    - $a$ before other threads have taken the lock
    - after
Mutual exclusion

\[ M_\Theta(r_{\gamma?}, o) = \Diamond \neq n(r, o) \rightarrow \gamma_c? \]
\[ M_\Theta(r_{\gamma?}, o) = \{} \]

\[ M_\Theta(r_{\gamma!}, o) = \gamma! \rightarrow \Box \neq n(r, o), \]
\[ \Diamond \neq n(r, o) \rightarrow \Box n(r_{\gamma!}, o) \]
data dependence

- jugment

\[ \vdash \Theta \ r : \gamma? \rightarrow d o \]

if \( o \in names(\gamma) \) and \( r' \gamma? \) is a prefix of \( r \).

- “\( o \) is potentially data-dependent on event/label \( \gamma? \) of trace \( r \)”

- note: it’s only potential dependence

\[
D_\Theta(r\gamma!) = \{ \gamma? \rightarrow \gamma! \} \quad \text{where} \quad \vdash \Theta \ \gamma? \rightarrow d fn(\gamma!) \cap \Delta(r)
\]

\[
D_\Theta(r\gamma?) = \{ \}
\]

For \( \Delta \), the definitions are applied dually.
control dependencies

- precedence nr. 3
- trivial

⇒ the events within each trace are linearly ordered
- notation

\[ \vdash a' \rightarrow^c a \]
putting it together: legal traces

- formal system to characterize interface behavior
- *non-branching* :-) 
- judgment:
  \[ \Xi; G \vdash r \triangleright s : \text{trace} \]
- “after \( r \) and with assumption/commitment-contexts \( \Xi \) and \( G \), the trace \( s \) is possible”
- context \( G \):
  - precedence graphs
  - cleanly separated into \( G_{\Delta} \) and \( G_{\Theta} \)
  - 3 reasons for precedence:
    1. \( \rightarrow^m \)
    2. \( \rightarrow^d \)
    3. \( \rightarrow^c \)
- \( G \) must remain acyclic: \( \vdash G \ ok \)
putting it together: legal traces

\[ \Xi \vdash r \triangleright o_s \xrightarrow{a} or \quad \Xi = \Xi + a \quad \Xi \vdash a : ok \]

\[ \hat{G}_\Theta = G_\Theta \cup G_\Theta(ra, or) \quad \hat{G}_\Delta = G_\Delta \cup G_\Delta(ra, os) \quad \vdash \hat{G}_\Delta : ok \]

\[ a = \nu(\Xi'). \ n\langle call \ or, l(\vec{v}) \rangle? \quad \Xi; \hat{G} \vdash r \ a \triangleright s : trace \]

\[ \Xi; G \vdash r \triangleright a \ s : trace \]

\[ \text{L-CALLI} \]
Results

- **Soundness** of the abstraction
- in particular: soundness of may and must:

**Lemma (Soundness of lock ownership)**

1. $\Xi \vdash C \xrightarrow{t} \Xi \vdash \dot{C}$ and $\Xi \vdash t : \Box_n o$, then thread $n$ has the lock of $o$ in $\dot{C}$.

2. If $\Xi \vdash C \xrightarrow{t}$ and $\Xi \vdash t : \Diamond_n o$ and there does not exist an $n' \neq n$ s.t. $\Xi \vdash t : \Box_{n'} o$, then $\Xi \vdash C \xrightarrow{t} \Xi \vdash \dot{C}$ for some $\Xi \vdash \dot{C}$ s.t. the thread $n$ has the lock of $o$ in $\dot{C}$.
introduction

semantics

interface description
  lock ownership
  data dependencies
  control dependencies

conclusion
Future work

- combination with cross-border instantiation/connectivity\(^3\)
- thread coordination:\(^4\)
  - wait
  - signal
- “cleaner” characterization:
  - non-determinism is theoretically (and practically) unpleasant
  - better: “real” strongest post-condition
  - “event-structures”?

\(^3\) conceptually not too complicated, technically tricky.
\(^4\) no ideas yet
References I

Abstract interface behavior of object-oriented languages with monitors.
Submitted as conference contribution.

A continuation proposal for cooperation between research groups in bilateral research program NWO/DFG,
May 2004.