Design issues in concurrent object-oriented languages and observability

Mai Thuong Tran and Martin Steffen
University of Oslo, Norway

International Conference on Knowledge and Systems Engineering,
Hanoi, Vietnam, October 14-17, 2011

What are we dealing with?

Effect of facets of object-oriented languages on the observable behavior of open programs

- Classes: units of code
- Inheritance: code re-use
- Concurrency: multi-threading and active objects
- Synchronization: locks and monitors

Why important?

- verification
- black-box testing
- compositionality, replacement, full abstraction

⟹ Easy question, difficult answer

⟹ Open semantics.

What are we dealing with?

Effect of facets of object-oriented languages on the observable behavior of open programs

- Classes: units of code
- Inheritance: code re-use
- Concurrency: multi-threading and active objects
- Synchronization: locks and monitors

Why important?

- verification
- black-box testing
- compositionality, replacement, full abstraction

⟹ Easy question, difficult answer

⟹ Open semantics.

What are we dealing with?

Effect of facets of object-oriented languages on the observable behavior of open programs

- Classes: units of code
- Inheritance: code re-use
- Concurrency: multi-threading and active objects
- Synchronization: locks and monitors

Why important?

- verification
- black-box testing
- compositionality, replacement, full abstraction

⟹ Easy question, difficult answer

⟹ Open semantics.

What are we dealing with?

Effect of facets of object-oriented languages on the observable behavior of open programs

- Classes: units of code
- Inheritance: code re-use
- Concurrency: multi-threading and active objects
- Synchronization: locks and monitors

Why important?

- verification
- black-box testing
- compositionality, replacement, full abstraction

⟹ Easy question, difficult answer

⟹ Open semantics.

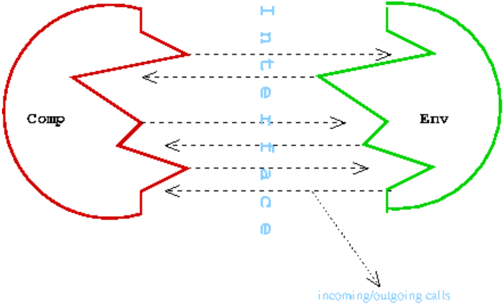
Notion of observation

```
public class C {  // component
    public static void main(String[] arg) {
        O x = new O();
        x.m(42);  // call to the instance of O
    }
}
```

```
class O {         // external  observer
    public void  m(int x) {
      ...
      System.out.println("success");
    }
}
```

Open systems

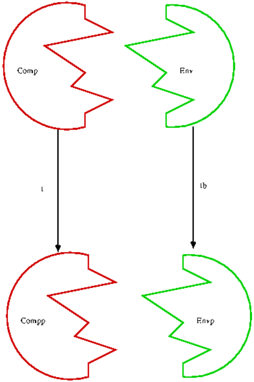
- Component = set of objects + threads “running” in parallel
- Environment = “context” = “observer”



- Component and its environment communicate via *method calls*.

Characterizing the open semantics

- "message passing"[1] framework ⇒ the corresponding open semantics is "traces" as interface interactions (method calls and returns)

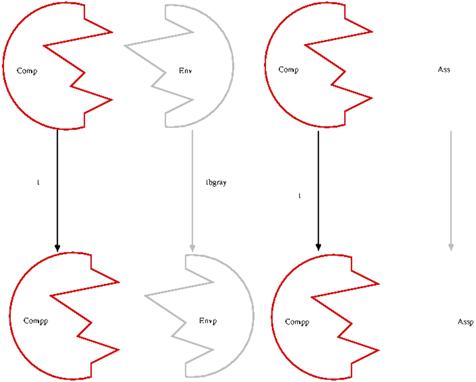


---

[1] no direct access to instance variables

Characterizing the open semantics

- "message passing"[1] framework ⇒ the corresponding open semantics is "traces" as interface interactions (method calls and returns)
- open = environment absent/arbitrary



---

[1] no direct access to instance variables

Characterizing the open semantics

- operational description: assumption/commitment formulation

$$Ass. \vdash C : Comm. \xrightarrow{a} \acute{Ass.} \vdash \acute{C} : \acute{Comm.} \tag{1}$$

- formal system to characterize interface behavior

$$\Delta \vdash C : \Theta \xrightarrow{a} \acute{\Delta} \vdash \acute{C} : \acute{\Theta} \ , \tag{2}$$

- interaction labels:

$$\begin{array}{lcll} \gamma & ::= & p\langle call\ o.l(\vec{v})\rangle \mid p\langle \text{return}(v)\rangle \mid \nu(n{:}T)_o & \text{basic labels} \\ a & ::= & \gamma? \mid \gamma! & \text{receive/send lab} \end{array}$$

Characterizing the observable behavior

- abstracting away the component, too:

$$\Delta, \Theta \vdash r \rhd t : \mathit{trace}$$

- inductive derivation system for legal traces:

$$\frac{\begin{array}{c}\text{check context: } \Delta, \Theta \vdash a \\ \text{update context: } \acute{\Delta}, \acute{\Theta} = \Delta, \Theta + a \\ \acute{\Delta}, \acute{\Theta} \vdash r\ a \rhd t : \mathit{trace} \qquad \text{other conditions}\end{array}}{\Delta, \Theta \vdash r \rhd a\ t : \mathit{trace}} \tag{3}$$

Classes?

what is the semantical import of classes?

1. interface separates observer and component classes

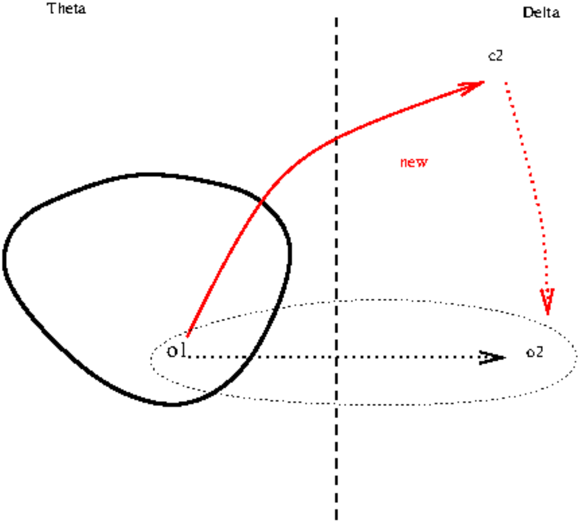
⇒ instantiation requests as interface interaction

2. class = generators of object (via `new`)[2]
3. abstraction of the heap topology

---

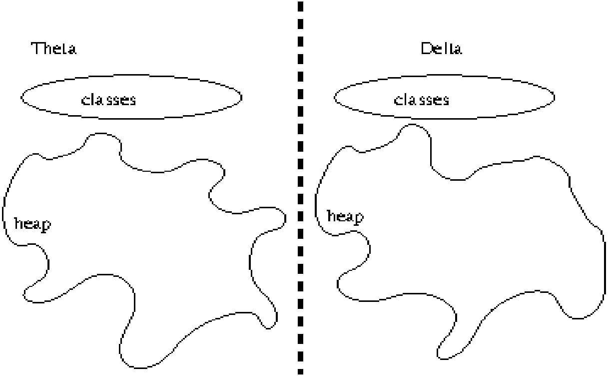
[2] Classes in *Java* or $C^\sharp$ serve also as kind of types, and furthermore for inheritance. We ignore that mostly here.

Cross-border instantiation & heap abstraction

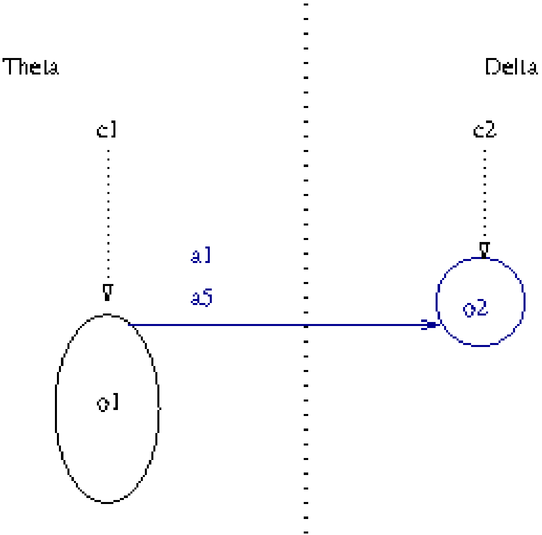


Heap separation

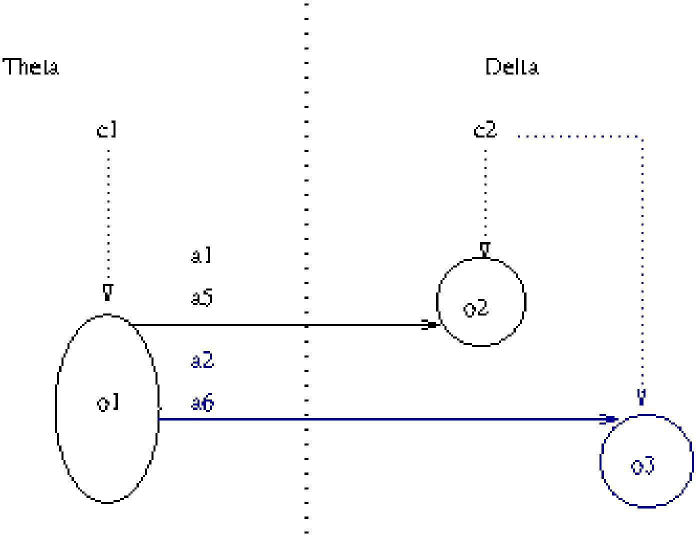
- heap is separated in component and environment part:



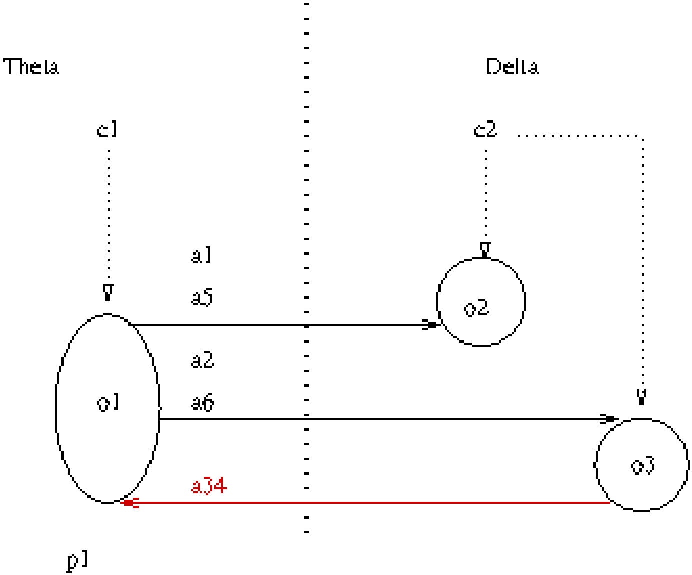
Dynamic heap abstraction example



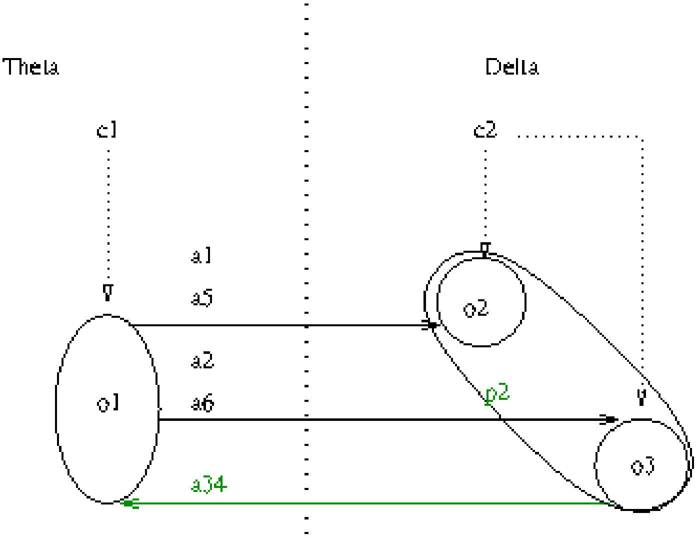
Dynamic heap abstraction: example



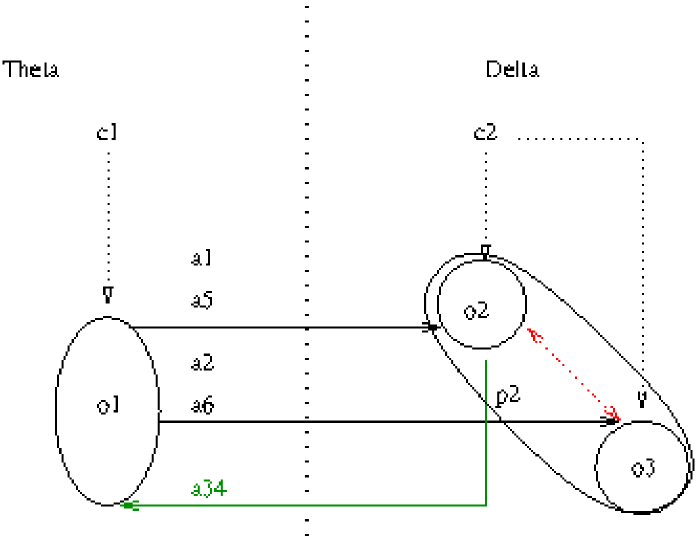
Dynamic heap abstraction: example



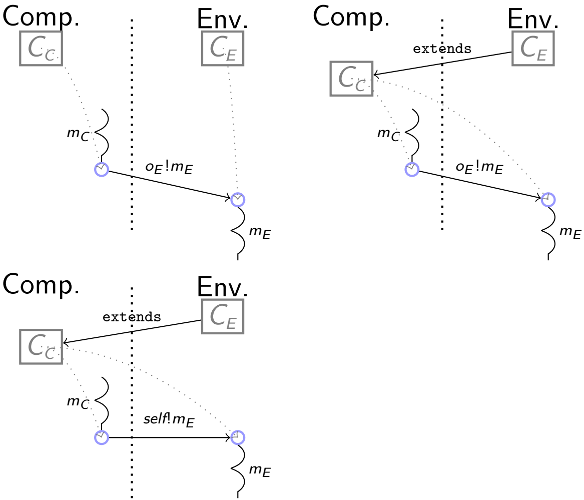
Dynamic heap abstraction: example



Dynamic heap abstraction: example



Cross-border inheritance



Consequences of inheritance

Separation in component & environment class + cross-border inheritance

1. self-calls observable.
2. abstraction of the heap topology
3. State of an object is split into two halves.

Synchronization

- shared (instance) state + concurrency ⇒ mutex
- sync. mechanism: monitors
- for instance in *Java*
- but: re-entrant monitors (recursion)

What changes?

- Now:

  The addition of monitors increase or decrease the discriminating power?

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

What changes?

- Now:

  > *The addition of monitors increase or decrease the discriminating power?*

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

What changes?

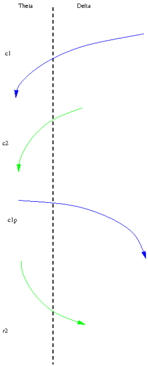
- Now:

  > *The addition of monitors increase or decrease the discriminating power?*

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

Example

- 2 calls, competing for the same (component) lock
- data dependence
  - o' received by the first call (of n_1)
  - returned by second thread n_2 afterwards
  - note: o' is **new**



- question: *is that trace possible?*

Example

- 2 calls, competing for the same (component) lock
- data dependence
  - o' received by the first call (of n_1)
  - returned by second thread n_2 afterwards
  - note: o' is **new**

$$\gamma_{c_1}?\ \gamma_{c_2}?\ \gamma'_{c_1}!\ \gamma_{r_2}! \qquad = $$

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

- question: *is that trace possible?*

Example

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

- question: *is that trace possible?*
- the answer is no!
- **data:** "n_1 before n_2"
- **monitors:**
  - the outgoing call of n_1 shows that n_1 must have the lock now

$\Rightarrow$ n_2 cannot have it now: $\Rightarrow$

"n_2 before n_1"

Example

$$\gamma_{c_1}?\ \gamma_{c_2}?\ \gamma'_{c_1}!\ \gamma_{r_2}! \quad = $$

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

- question: *is that trace possible?*

$\gamma_{c_1}?$ $\gamma_{c_2}?$

(2)

<u>Note:</u> *non-atomic* lock-grabbing ⇒ **no order!**

Example

$$\gamma_{c_1}?\ \gamma_{c_2}?\ \gamma'_{c_1}!\ \gamma_{r_2}! \qquad = \qquad (\nu o':c)\, n_1\langle \mathit{call}\ o.l(o')\rangle ?\ n_2\langle \mathit{call}\ o.l()\rangle ?\ n_1\langle \mathit{call}\ \tilde{o}.l()\rangle !\ n_2\langle \mathit{return}(o')\rangle !$$

- question: *is that trace possible?*

$$\begin{tikzcd}
\gamma_{c_1}? \arrow[d, "n_1"] & \gamma_{c_2}? \\
\gamma'_{c_1}! &
\end{tikzcd}$$

(3)

Note: *there is* no *order between events of* n_1 *and* n_2*!*

Example

$$\gamma_{c_1}?\ \gamma_{c_2}?\ \gamma'_{c_1}!\ \gamma_{r_2}! \qquad = $$

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

- question: *is that trace possible?*

$$\begin{tikzcd}
\gamma_{c_1}? \arrow[d, "n_1"'] \arrow[rd, bend left, "o'"] & \gamma_{c_2}? \arrow[d, "n_2"] \\
\gamma'_{c_1}! & \gamma_{r_2}! \arrow[lu, bend left]
\end{tikzcd} \qquad (4)$$

Note:

- data dependence because of o'

Active object

- active objects
  - aynchronous method calls ⇒ each method call = new thread
  - no re-entrance
  - unit of “state” ⇒ unit of concurrency
- lock state not observable
- observable semantics *much* easier
- better compositionality

Conclusions

- Object-orientation and modularity.
- Concurrency.
- Synchronization.

References I

[Ábrahám et al., 2008a] Ábrahám, E., Grüner, A., and Steffen, M. (2008a).
Abstract interface behavior of object-oriented languages with monitors.
Theory of Computing Systems, 43(3-4):322–361 (40 pages).

[Ábrahám et al., 2008b] Ábrahám, E., Grüner, A., and Steffen, M. (2008b).
Heap-abstraction for an object-oriented calculus with thread classes.
Journal of Software and Systems Modelling (SoSyM), 7(2):177–208 (32 pages).

[Steffen, 2006] Steffen, M. (2006).
Object-Connectivity and Observability for Class-Based, Object-Oriented Languages.
Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel.
281 pages.