Object Models in the πL -Calculus

Jean-Guy Schneider

Software Composition Group
Institute of Computer Science and Applied Mathematics (IAM)
University of Berne

E-mail: `schneidr@iam.unibe.ch`
WWW: `http://www.iam.unibe.ch/~schneidr/`

Overview

- πL-calculus based object model
- integration of Generic Synchronization Policies
- pre-methods, generators
- class abstractions
- inheritance, method dispatch strategies
- mixins
- references

Pierce/Turner Basic Object Model

The basic object model of Pierce and Turner captures the essential features of objects:

def emptyRef (X) = $(\upsilon\ c, s, g)$

$$
\begin{array}{ll}
(& \overline{X_{reply}}(\langle set=s, get=g \rangle) \\
| & !g(Y).c(Z).(\ \overline{Y_{reply}}(Z) \mid \bar{c}(Z)\ ) \\
| & s(Y).\ (\ \ \bar{c}(Y) \mid \overline{Y_{reply}}(\langle\rangle) \\
  & \qquad\quad \mid\ !s(Z).c(V).(\ \bar{c}(Z) \mid \overline{Z_{reply}}(\langle\rangle)\ ) \\
  & \qquad\quad) \\
) &
\end{array}
$$

Objects in the π-Calculus

Sangiorgi's translation of an untyped OC(Adabí/Cardelli) into the polyadic π-calculus:

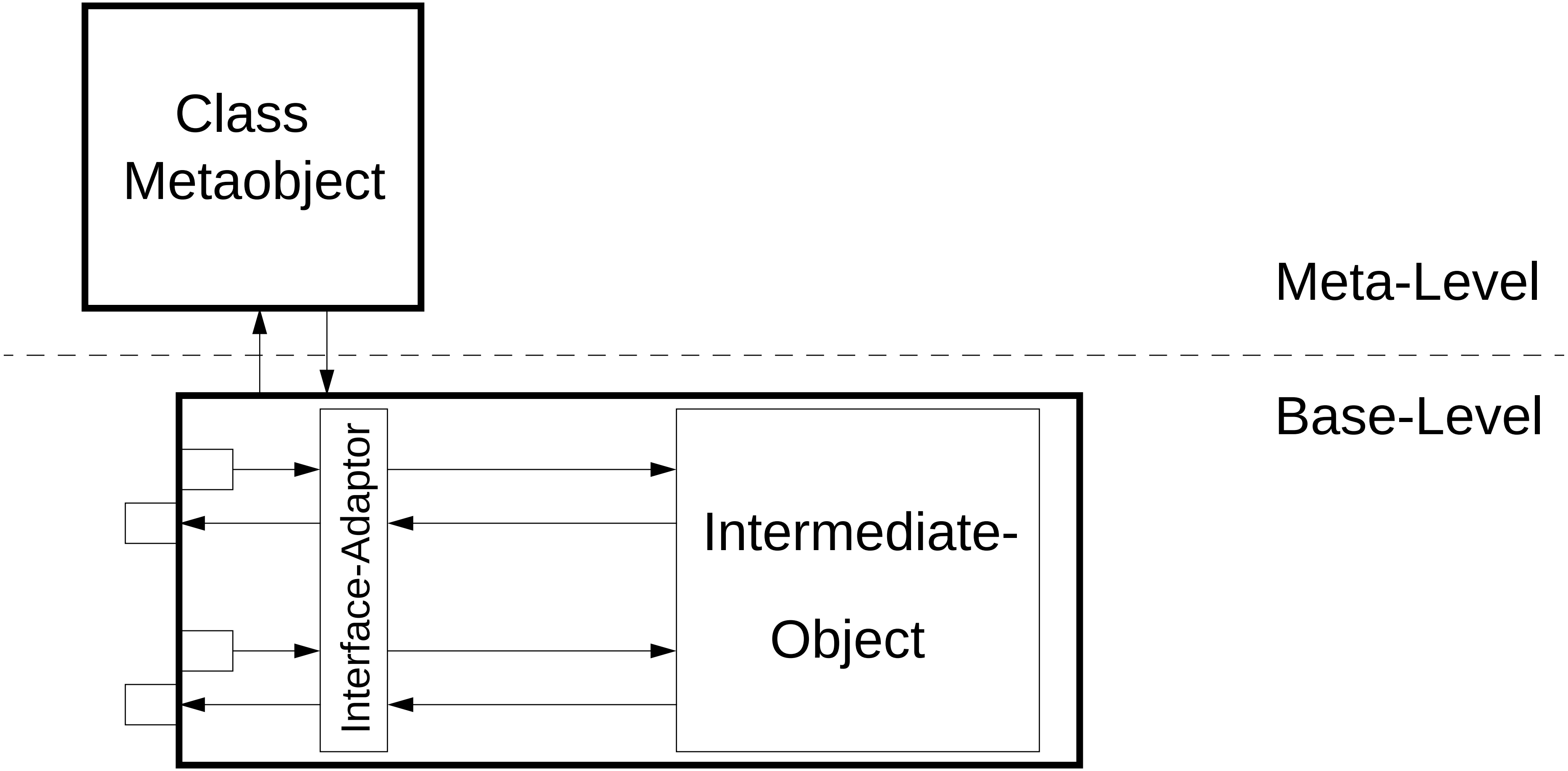
$$[\{_{j\in 1..n}\ l_j = \zeta(y).b_j\}]_p \quad =^{def} \quad \overline{p}(x).!x(l,r,y).(\ \Pi_{j\in 1..n}[l = l_j]\ [b_j]_r\ )$$

$$[a.l_j]_p \quad =^{def} \quad (\upsilon\ q)(\ [a]_q \mid q(x).\overline{x}\langle l_j,p,x\rangle\ )$$

$$[a.l_j \Leftarrow \zeta(y).b]_p \quad =^{def} \quad (\upsilon\ q)(\ [a]_q \mid q(x).\overline{p}(x_{new}).!x_{new}(l,r,y).\ (\ [l = l_j][b]_r \mid [l \neq l_j]\overline{x}\langle l,r,y\rangle\ )$$

$$[x]_p \quad =^{def} \quad \overline{p}x$$

πL-Calculus based Object Model (I)



πL-Calculus based Object Model (II)

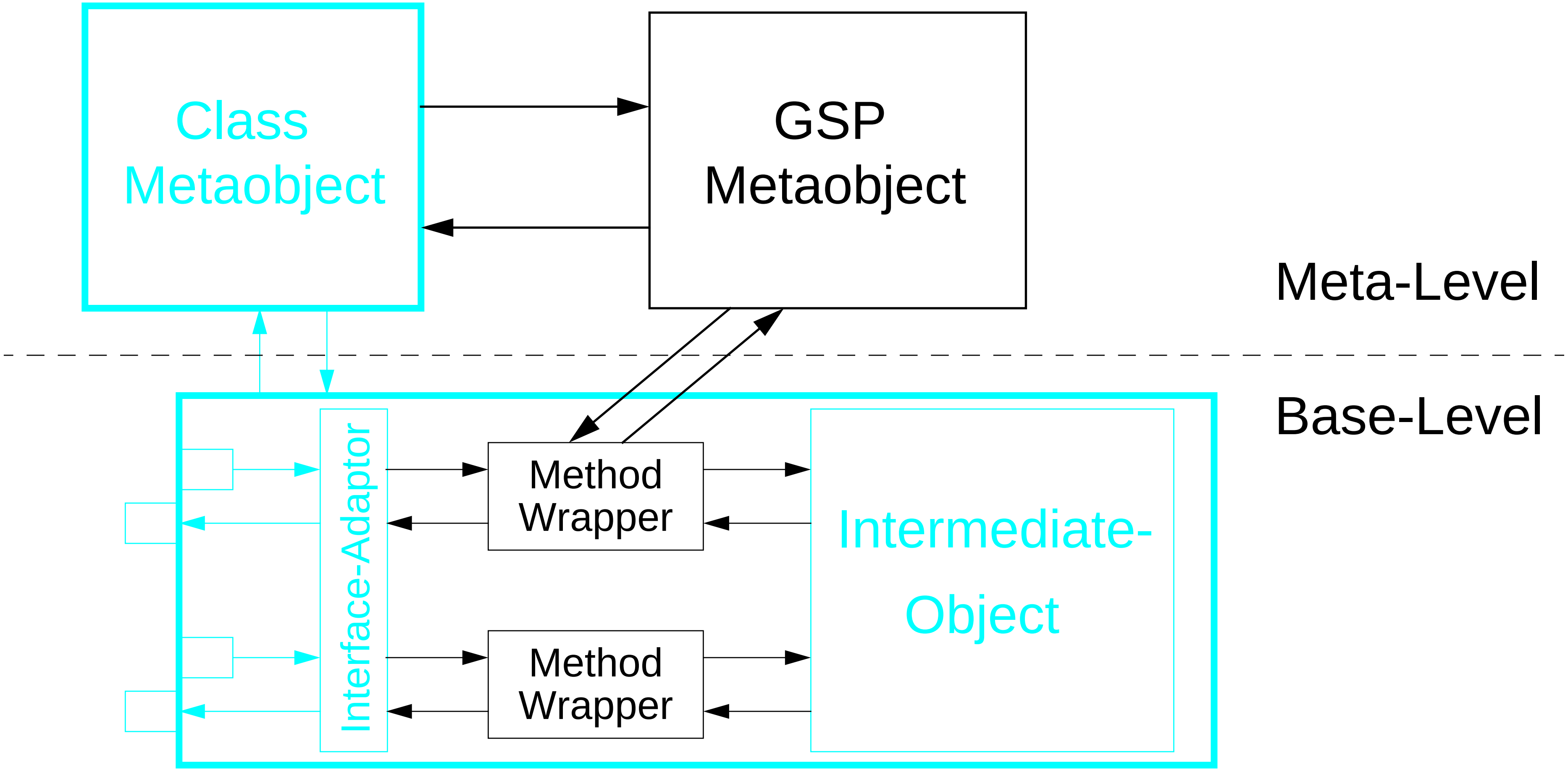
$$[\{_{j \in 1..n}\ l_j = \zeta(y).b_j\}]_r \quad =^{def} \quad (\upsilon\ x_1,\ldots,x_n)(\ \bar{r}(<l_1{=}x_1,\ldots,l_j{=}x_j>) \ \mid \Pi_{j \in 1..n}\ !x_j(X).[b_j]_{X_{reply}}\ )$$

$$[\{_{j \in 1..n}\ l_j = b_j\}]_r \quad =^{def} \quad (\upsilon\ x_1,\ldots,x_n,s,t)([\{_{j \in 1..n}\ \underline{l_j =}\ \zeta(y).b_j\}]_t \ \mid t(S).(\bar{r}(S) \mid !s(Y).\ X_{reply}(S) \ \mid \Pi_{j \in 1..n}\ !x_j(X).[S.l_j(<X,self{=}s>)]\ ))$$

$$[O.l_j(X)]_r \quad =^{def} \quad (\upsilon\ p)([O]_p \mid p(Y).\overline{Y_{l_j}}(<X,reply{=}r>)\ )$$

$$[F]_r \quad =^{def} \quad \bar{r}(F)$$

Integration of GSP's into Object Model



Observations

- record-based basic object model is a robust basis for modelling object-oriented features,
- intermediate-objects as collections of <u>pre-methods</u>,
- controlling visibility of features based on scoping rules,
- classes as first class entities: <u>class metaobjects</u>,
- inheritance as intermediate-object <u>extension</u>,
- π–calculus expressive enough to model common features of OOPL’s.

☞ Problem: cannot define reusable <u>class abstractions</u> due to the usage of pre-methods (explicit *self*-binding).

From Pre-methods to Generators

Generator:

- defines behaviour of objects,
- requires *self* as additional parameter,
- Δ defines *difference* in relation to a parent class.

$$G_D(self) = G_P(self) \oplus \Delta(self, G_P(self))$$

Wrapper:

- *fixed-point operator* over a generator,
- establishes correct *self-binding*.

$$W = fix_{self}[G(self)]$$

☞ Inheritance as <u>generator composition</u>

Class Abstractions

A class abstraction (i.e. a *function*):

- defines a class metaobject
- requires a Δ and a reference to a parent-class metaobject

$$C = class(\Delta, parent)$$

Generator composition defines the inheritance model of a class:

$$G_D(self) = G_P(self) \oplus \Delta(self, G_P(self))$$

$$G_B(self, l) = \Delta(self, l) \oplus G_P(self, l \oplus \Delta(self, l))$$

Application of fixed-point operator defines method dispatch:

$$G_S(self) = fix_{self'}[G_P(self') \oplus \Delta(self', G_P(self'))]$$

Encoding of the Fixed-point Operator

The encoding of the fixed-point operator is based on a *reference cell* and *self* being a function (and not a value):

$$
\begin{array}{l}
\textbf{def}\ \mathrm{wrapper}(\mathrm{Init},\mathrm{res}) = (\upsilon\ r, s, x)\ (\ \overline{\mathrm{emptyRef}}(\langle \mathrm{reply}=x\rangle) \\
\qquad\qquad\qquad\qquad |\ \ x(S).(\ \ !s(X).\overline{S_{get}}(X) \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \ \overline{\mathrm{generate}}(\langle \mathrm{init}=\mathrm{Init},\mathrm{self}=s,\mathrm{reply}=r\rangle) \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \ r(Y).(\ \overline{S_{set}}(Y)\ |\ \overline{\mathrm{res}}(Y)\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad) \\
\qquad\qquad\qquad\qquad)
\end{array}
$$

☞ functions are encoded as <u>replicated processes</u>

Mixins

A mixin is an *abstract subclass* (a “subclass” without specified parent-class):

$$G_M(self, G_P) = G_P(self) \oplus \Delta(self, G_P(self))$$

Applying a mixin *M* to a class *C* merges the behaviour of *M* and *C*:

$$G_{M \bullet C}(self) = G_C(self) \oplus G_M(self, G_C)$$

$$W_{M \bullet C} = fix_{self}[G_{M \bullet C}(self)]$$

Mixin composition:

$$G_{M_{1,2}}(self, G_P) = G_P(self) \oplus G_{M_1}(self, G_P) \oplus G_{M_2}(self, G_P)$$

☞ Mixin composition/application is associative

Summary

- an object is viewed as an agent containing local channels (representing state) and agents (representing behaviour),
- class and mixin abstractions as functions; classes and mixins as meta-level objects,
- subclass specification based on incremental derivation,
- *self*-binding and method dispatch strategies based on fixed-point operators,
- compositional view of object-oriented abstractions (e.g., inheritance as composition of generators):

☞ Unifying concept of agents and forms

References

- Benjamin Pierce and David Turner. *Concurrent Objects in a Process Calculus,* 1995.
- Ciaran McHale. *Synchronization in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*, 1994.
- Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. *Using Metaobjects to Model Concurrent Objects in PICT*, 1996.
- Jean-Guy Schneider and Markus Lumpe. *Synchronizing Concurrent Objects in the π-Calculus*, 1997.
- Jean-Guy Schneider. *Component, Scripts, and Glue: A Conceptual Framework for Software Composition*, 1999.