A Metamodel for Concurrent, Object-based Programming[*]

Jean-Guy Schneider — **Markus Lumpe**

Software Composition Group,
Institut für Informatik (IAM), Universität Bern
Neubrückstrasse 10, CH-3012 Bern, Switzerland
Tel: +41 31 631 48 68, Fax: +41 31 631 39 65

{schneidr,lumpe}@iam.unibe.ch

ABSTRACT. *The development of flexible and reusable concurrent object-oriented programming abstractions has suffered from the inherent problem that reusability and extensibility is limited due to position-dependent parameters. To tackle this problem, we propose the* FORM-*calculus, an inherently polymorphic variant of the π-calculus, where polyadic tuple communication is replaced by monadic communication of extensible records. This approach facilitates the specification of flexible, concurrent, object-oriented programming abstractions. Based on our previous work in this field, we present a* FORM-*calculus based meta-level approach for concurrent, object-based programming which adapts a compositional view of programming. This approach enables the definition of various semantic models supporting different kinds of inheritance and method dispatch strategies, and clarifies concepts which are typically merged in existing object-oriented programming languages.*

RÉSUMÉ. *Le développement d'abstractions objets concurrentes, flexibles et réutilisables a souffert du problème suivant: la réutilisabilité et l'extensibilité sont limitées par le fait que les paramétres ont une position fixe. Pour résoudre ce problème, nous proposons le* FORM-*calculus, une version polymorphic du π-calculus, dans laquelle la communication polyadique basée sur des tuples est remplacée par une communication monadique d'enregistrements extensibles. Cette approche facilite la spécification d'abstractions orientées-objets flexibles et concurrentes. A partir de nos précédents travaux, nous présentons une approche méta basée sur le* FORM-*calculus qui propose une vue compositionnelle de la programmation. Cette approche permet la définition de différents modèles sémantiques de l'héritage et des recherches de méthodes clarifiant ainsi des concepts souvent implicites dans les langages objets traditionnels.*

KEYWORDS: *π-calculus, object models, meta-level framework, asymmetric record concatenation.*

MOTS-CLÉS : *π-calculus, modèles objets, cadre de conception méta, concaténation assymétrique d'enregistrements.*

---

1. Introduction

Concurrent programming demands special languages that provide primitives for communication and synchronization. Using objects it turns out that a general programming model can easily be made concurrent and parallel. Concurrent object-based programming language development, however, has long suffered from the lack of any generally accepted formal foundation for defining their semantics.

Several authors have shown that the π-calculus, or some of its variants [MIL 90, MIL 91, HON 92], are expressive enough for modelling standard object-oriented programming language features in a convenient way and may therefore serve as a suitable semantic foundation for defining a concurrent object-oriented language. For example, Walker [WAL 95] has shown that POOL [AME 86] can be modelled in the π-calculus, Vasconcelos has shown that subtyping and a notion of self can be modelled with the "Calculus of Objects" [VAS 94], and Barrio has given a nearly complete representation of active objects in the π-calculus [BAR 95].

Using the pure π-calculus, objects can be viewed as groups of processes. There is, however, no way that one process can directly affect or refer to another process. A process can only send messages along some channels where, by convention, the other process listens. Similarly, referring to a group of processes means that we can send messages to a collection of channels where these processes are listening. An attractive notion to model such a facility is the standard notion of *records* that enable us to selectively address one member of a group by using its *name*. Viewing each individual channel as an explicitly named "service access point", we can bundle them together in a record that provides a well-defined interface for accessing the related services. Furthermore, this packaging gives rise to a *higher-order* style of programming with objects, since a complete interface of one object may be manipulated as a single value.

Motivated by the basic object model of Pierce and Turner [PIE 95], we have previously defined a record-based object model in the polyadic π-calculus [MIL 91] that incorporates known languages features of object-oriented programming languages such as encapsulation, object identity, instantiation, synchronization, dynamic binding, inheritance, overriding, and class variables [LUM 96, SCH 97]. In this model, classes are represented as runtime entities (i.e. as *class metaobjects*), which turned out to be useful for (i) the declaration, initialization, and access control of class variables, (ii) the implementation of class methods, (iii) the creation and initialization of instances, and (iv) modelling inheritance.

This model, however, did not allow us to explore the common behaviour and protocols of class metaobjects: each class metaobject had to be created from scratch, hardcoding both the underlying inheritance model and method-dispatch strategy. Furthermore, using the polyadic π-calculus, there exists the inherent problem that reusability and extensibility of abstractions are limited due to position-dependent parameters. For example, the specification of generic readers/writers synchronization policies cannot be directly coded without wrapping method arguments in order to treat an arbitrary number of arguments as a single value [SCH 97].

In order to address these problems, we have defined the FORM-calculus, an inherently polymorphic variant of the π-calculus, in which agents communicate by passing *forms* (a special notion of extensible records) rather than tuples. Besides forms, which have their analogues in many existing programming languages and systems (e.g., HTML, Visual Basic, Python), the FORM-calculus incorporates also *polymorphic form extension* and *polymorphic form restriction* as basic operations on forms. Polymorphic form extension is a concept that corresponds to asymmetric record concatenation [CAR 94] while polymorphic form restriction can be considered as the corresponding "inverse" operation.

Forms, polymorphic extension, and polymorphic restriction, we argue, are the key mechanisms for extensibility, flexibility, and robustness as (i) clients and servers are freed from fixed, positional tuple-based interfaces, (ii) abstractions are more naturally polymorphic as interfaces can be easily extended, and (iii) environmental arguments (such as communication policies or default I/O-services) can be passed implicitly. In particular, polymorphic extension and polymorphic restriction in combination with keyword-based parameters facilitate the definition of object-oriented abstractions as they enable the definition of an extensible and reusable meta-level framework for modeling concurrent object-oriented features. Furthermore, by using a process calculus, concurrency does not need to be modeled explicitly, and we can focus on the definition of extensible, higher-level abstractions.

This paper is organized as follows: in section 2, we informally introduce key concepts of the FORM-calculus which forms the formal basis of this work. In section 3, we define our previous object model in terms of the FORM-calculus and identify the key concepts for extension and generalization. Driven by the results of the previous section, we define a meta-level framework for modeling concurrent object-oriented programming abstractions in section 4. We conclude with a summary of the main observations and a discussion about related work in section 5.

2. Key concepts of the FORM-calculus

In this section, we informally introduce the key concepts of the FORM-calculus [SCH 99]. We will not present, however, the complete definition of the calculus, since this is beyond the scope of this paper.

The FORM-calculus is an inherently polymorphic variant of the (polyadic) π-calculus [BOU 92, MIL 92, HON 92, SAN 95], where the communication of tuples is replaced by communication of so-called *forms*, a special notion of extensible records. More precisely, in the FORM-calculus polyadic tuple communication is replaced by monadic communication of forms. By this approach, we address the inherent problem that reusability and extensibility is limited due to position-dependent parameters.

Forms are finite mappings from an infinite set $\mathcal{L}$ of labels to an infinite set $\mathcal{N}^+ = \mathcal{N} \cup \{\mathcal{E}\}$, the set of names extended by $\mathcal{E}$ that denotes the empty binding. The basic operations on forms are *binding extension*, *polymorphic extension*, and *polymorphic*

$$
\begin{array}{lll}
A & ::= & \mathbf{0} \qquad\quad\ \textit{inactive agent} \\
& | & A \mid A \qquad \textit{parallel composition} \\
& | & (\nu\, a)A \qquad \textit{restriction} \\
& | & V(X).A \qquad \textit{input} \\
& | & !V(X).A \qquad \textit{replication} \\
& | & \overline{V}(F) \qquad\ \textit{output}
\end{array}
$$

$$
\begin{array}{lll}
F & ::= & \langle\rangle \qquad\quad \textit{empty form} \\
& | & F\langle l{=}V\rangle \quad \textit{binding extension} \\
& | & X \qquad\quad\ \textit{form variable} \\
& | & F{\cdot}X \qquad\ \textit{polym. extension} \\
& | & F\backslash X \qquad\ \textit{polym. restriction} \\
\\
V & ::= & \mathcal{E} \qquad\quad\ \textit{empty binding} \\
& | & a \qquad\quad\ \textit{simple name} \\
& | & X_l \qquad\quad \textit{projection}
\end{array}
$$

Table 1: Syntax of FORM-calculus agents and forms.

restriction (see Table 1). A binding extension $F\langle l = V\rangle$ adds a binding for label l with value V to a given form F. If F already defines a binding for label l, then the corresponding value is overridden. Polymorphic form extension is a concept that corresponds to asymmetric record concatenation [CAR 94]. Polymorphic restriction is an operation which removes the set of labels from a given form.

The FORM-calculus supports similar operators as the π-calculus. $\mathbf{0}$ denotes the inactive agent. Parallel composition runs two agents in parallel. The restriction $(\nu\, a)A$ creates a *fresh* name a with scope A. An input-prefixed agent $V(X).A$ waits for a form F to be sent along the channel denoted by value V and then behaves like $A\{F/X\}$, where $\{F/X\}$ is the substitution of all form variables X with form F. An output $\overline{V}(F)$ emits a form F along the channel denoted by value V. A replication $!V(X).A$ stands for a countably infinite number of copies of $V(X).A$ in parallel.

The operational semantics of the FORM-calculus is given using the reduction system technique proposed by Milner [MIL 90], where axioms for a structural congruence relation are introduced prior to the definition of the reduction relation. The axioms for structural congruence and the reduction relation of the FORM-calculus are similar to the ones of the π-calculus. Note, however, that reduction is only defined on closed agents (i.e. agents not containing free form variables) and agents like $\mathcal{E}(X).A$ or $\overline{\mathcal{E}}(F)$ are identified with the inactive agent $\mathbf{0}$ (refer to [SCH 99] for details).

In the FORM-calculus, form variables are *polymorphic placeholders* for form values. In order to access elements of a form, we use *name projections*. In fact, projections denote keyword-based *named formal process parameters*. A projection X_l has to be read as selection of the parameter named by l. The reader should note that projection is only defined on *closed* forms, i.e., forms that do not contain any form variable. Given two (closed) forms F and G, name projection is defined as shown in Table 2. Note that a form may have multiple bindings for label l. In this case, name projection ensures that a name projection always extracts the *rightmost binding*.

$$
\begin{array}{rcl}
\langle\rangle_l & = & \mathcal{E} \\
(F\langle l\!=\!x\rangle)_l & = & x \\
(F\langle m\!=\!y\rangle)_l & = & F_l, \quad \text{if } m \neq l
\end{array}
\qquad
\begin{array}{rcl}
(F\cdot G)_l & = & \left\{ \begin{array}{ll} F_l, & \textit{if } G_l = \mathcal{E} \\ G_l, & \textit{otherwise} \end{array} \right. \\
(F\backslash G)_l & = & \left\{ \begin{array}{ll} F_l, & \textit{if } G_l = \mathcal{E} \\ \mathcal{E} & \textit{otherwise} \end{array} \right.
\end{array}
$$

Table 2: Name projection.

Now, in order to formalize the description of the behaviour of object-oriented abstractions, we will use the following notations that facilitate the presentation of our meta-level framework. These notations can be though of as syntactic sugar on top of the FORM-calculus, i.e., there exists a sound interpretation of them in terms of the primitives of the calculus (e.g., a function is represented as a replicated input agent) [TUR 96, LUM 99, SCH 99].

The expression $\{m_1(X_1) \rightarrow b_1, \ldots, m_n(X_n) \rightarrow b_n\}$ denotes a form with fields $m_1, ..., m_n$ representing methods with the method bodies $b_1, ..., b_n$ and the (keyword-based) formal arguments $X_1, ..., X_n$, respectively. In fact, this expression stands for an interface of an active object (i.e. it is represented by a FORM-calculus agent). The selection of a component a of a form value r is denoted by $r.a$. The update of a field l with value v is denoted by $\{l = v\}$. We use the operators $\oplus$ and $\backslash$ to denote the polymorphic extension and polymorphic restriction of two forms, respectively. Finally, we use "let $V = e_1$ in e_2" to assign expression e_1 to V in e_2 and $fix_X[f(X)]$ to denote the least fixed-point of a function f.

3. Towards a common metamodel

The central idea of the object model presented in [LUM 96, SCH 97] is to represent classes as first-class entities. Using this approach, we can incorporate various abstractions found in object-oriented programming, but the inherent problem of this approach is the limited reusability and extensibility due to position dependent parameters at both the base level and the metaobject level. In order to overcome these shortcomings, we proceed as follows: (i) we use the newly defined FORM-calculus as formal basis for modelling object-oriented abstractions and (ii) we define a generalization of our former approach based on the proposals made by Bracha and Cook [BRA 90], Cook and Palsberg [COO 94], Van Limberghen and Mens [VAN 96], and Rossie et. al [ROS 96a].

Cook and Palsberg [COO 94] have proposed an approach for modelling classes, mixins, inheritance etc. using the notion of *generators* and *wrappers*. A generator, denoted by G, defines the behaviour of objects (encoded as records) with an unbound self-reference whereas a wrapper, denoted by W, establishes the correct binding of self. In fact, a generator is a *lambda abstraction* over self (i.e. a function that requires

a parameter for self) and a wrapper is the *fixed-point operator* for the corresponding generator. A similar approach has also been proposed by Reddy, but the explicit separation between generators and wrappers is omitted [RED 88].

Using the approach of Bracha and Cook [BRA 90], we define the generator for a class as the composition of its parent-class generator (or a collection of parent-class generators in case of multiple derivation) and an *incremental modification*, written Δ. Furthermore, we shall use the term *intermediate object* to denote both the result of a generator and Δ since both abstractions yield objects with unbound self-references.

In order to illustrate this approach, consider the Java specification of the class Point shown in Figure 1. This class defines two private instance variables x and y, two public functions *X* and *Y* to access the values of the instance variables, two public methods *move* and *double* to change the values of the x- and y-coordinates of a Point instance, and a constructor *Point* to correctly initialize the private instance variables.

Using Δ_{Point} to stand for the incremental modification defined by class Point, the generator G_{Point}, the wrapper W_{Point}, and I denoting the constructor arguments, the class *Point* is defined as follows:

$$
\begin{aligned}
\Delta_{Point}(I) \quad = \quad & \text{let } x = I.ix, \ y = I.iy \\
& \text{in} \\
& \qquad \{ \ X() \to x, \ Y() \to y, \\
& \qquad \quad move(Args) \to x = x + Args.dx; \ y = y + Args.dy, \\
& \qquad \quad double() \to I.self.move(\{dx = x, dy = y\}) \ \} \\
G_{Point}(I) \quad = \quad & \Delta_{Point}(I) \\
W_{Point}(I) \quad = \quad & fix_s \, [\, G_{Point} \, (I \oplus \{self = s\}) \,] \\
Point \quad = \quad & \{ W(I) \to W_{Point}(I), \ G(I) \to G_{Point}(I) \}
\end{aligned}
$$

The expression $fix_s \, [\, G_{Point}(I \oplus \{self = s\})]$ yields an object with an appropriately bound self-reference, which is expressed by the binding $self = s$. The class $Point$ is represented as a form which defines bindings for its generator and wrapper. In order to create an instance of class *Point*, one needs to call the method W with appropriate constructor arguments (i.e. W is in fact the constructor of class $Point$).

The reader should note that instance variables, methods, and object instances as well as Δ_{Point}, G_{Point}, and W_{Point} are modelled by FORM-calculus-agents. Therefore, the whole system is concurrent. More precisely, when the system is executed, it evolves by means of communicating agents. The chosen notation, however, allows us to smoothly omit this fact and to focus on the modelling of object-oriented abstractions. In fact, using the FORM-calculus, we get concurrency by default, but in general not for free (e.g., special synchronization schemes need to be specified explicitly).

A specialization of the class Point can be defined the same way. Suppose we want that the y coordinate of an instance never exceeds a given upper bound b. The corresponding class is called BoundedPoint and the incremental modification Δ_{BPoint}, generator G_{BPoint}, and wrapper W_{BPoint} are defined as follows:

```
class Point {
      private int x;
      private int y;

      public Point (int ix, int iy) { x = ix; y = iy; }
      public int X() { return x; }
      public int Y() { return y; }
      public void move (int dx, int dy) { x = x + dx; y = y + dy; }
      public void double () { move (x, y); }
}

class BoundedPoint extends Point {
      private int b;

      public BoundedPoint (int ix, int iy, int ib) { b = ib; super (ix, iy); }
      public int bound() { return b; }
      public void move (int dx, int dy) {
            if (this.Y()+dy < this.bound()) { super.move (dx, dy); } }
}
```

Figure 1: Java code of the classes Point and BoundedPoint.

$$
\begin{aligned}
\Delta_{BPoint}(I) \quad &= \quad \text{let } b = I.ib \\
&\qquad \text{in} \\
&\qquad\qquad \{\ bound() \to b, \\
&\qquad\qquad\quad move(A) \to if\ (I.self.Y()+A.dy) < I.self.bound() \\
&\qquad\qquad\qquad\qquad\qquad\qquad then\ I.orig.move(A)\ \} \\
G_{BPoint}(I) \quad &= \quad \text{let } Obj_I = Point.G(I) \\
&\qquad \text{in} \\
&\qquad\qquad Obj_I\ \oplus\ \Delta_{BPoint}\ (I \oplus \{orig = Obj_I\}) \\
W_{BPoint}(I) \quad &= \quad fix_s\ [\ G_{BPoint}\ (I \oplus \{self = s\})\ ] \\
BoundedPoint \quad &= \quad \{W(I) \to W_{BPoint}(I),\ G(I) \to G_{BPoint}(I)\}
\end{aligned}
$$

In order to correctly dispatch self-calls and to have a sound interpretation of the inherited parent behaviour in class BoundedPoint, the abstraction Δ_{BPoint} requires two parameters: $self$ (the argument I contains a binding for $self$) and $orig$ that gives access to the inherited behaviour. Furthermore, the argument I passed to Δ_{BPoint} also provides bindings for the initial values used in the constructor (e.g., ix, iy, and ib). Note that (i) the generator G_{BPoint} defines a composition of the intermediate object Obj_I (instantiated by the generator G_{Point}) and the intermediate object generated by Δ_{BPoint} and (ii) the application of the fixed-point operator in W_{BPoint} on G_{BPoint} (i.e. the composition of both intermediate objects) ensures a correct binding of $self$ within the resulting object [LUM 96].

Analyzing the examples, we can see that the wrappers W_{Point} and W_{BPoint} are almost identical: they only differ in the used generator. Furthermore, the composition of the two intermediate objects in G_{BPoint} is very similar to the way inheritance is defined in Smalltalk [GOL 89], and it can be assumed that the underlying inheritance mechanism does not differ in the same semantic model. Hence, by appropriately parameterizing the corresponding generators and wrappers, the same G and W abstractions can be used for different classes. Summarizing this observation, we argue that generators and wrappers define the *semantic model* (i.e. the underlying inheritance model, the method-dispatch strategy etc.) for a related family of classes whereas Δ and the collection of parent-class generators specify the behaviour of a concrete class.

Further analyzing the concept of generators and wrappers, we can see that they also specify a *meta-level protocol*: a generator, for example, creates intermediate object(s) of its parent-class(es) and the intermediate object of Δ, and composes these objects according to the given semantic model. The way these intermediate objects are composed is different for each semantic model (e.g., in a Smalltalk-like model, methods of a class have precedence over the methods defined in a parent-class whereas in a Beta-style model [MAD 93], the parent-class methods have precedence), but the "ingredients" of the composition are the same for all semantic models. Hence, it is possible to split the functionality of generators into a static protocol-part and a variable model-part, denoted as *concrete* and *model* generators, respectively. Therefore, it is sufficient to define only one concrete generator and parameterize it with appropriate model generators. A similar separation can also be achieved for wrappers.

4. A meta-level framework

Driven by our motivation to generalize the definition of concurrent, object-oriented abstractions and the observations described in the previous section, we have defined the meta-level framework for modeling concurrent, object-oriented programming abstractions illustrated in Figure 2. This framework defines a hierarchy of meta-level abstractions. Arrows between entities at different levels denote inter-level dependencies whereas the annotation on the arrows represent the formal arguments of the corresponding dependencies.

On top of the hierarchy we have the abstraction MetaModel which defines the generic framework for meta-class and class abstractions. It provides the common behaviour of all meta-class abstractions, the corresponding meta-protocol, and in particular the concrete generators, wrappers, and composers. The reader should note that a composer abstraction is solely used for mixin application and composition (discussed in section 4.3). In fact, the abstraction MetaModel is a *meta-meta-class* abstraction.

The meta-class abstractions Class, Mixin, and Encapsulate define the meta-behaviour for a specific semantic model and are instantiated by passing appropriate model generators, model wrappers, and model composers to the abstraction MetaModel.
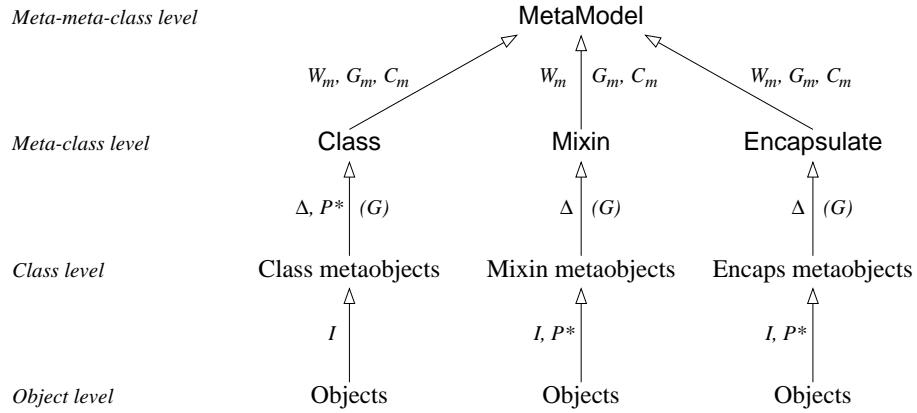
Figure 2: Conceptual view of meta-level framework.

Like in the object model presented in [LUM 96, SCH 97], classes are represented as *class metaobjects*. The extended model differs with respect to the previous approach that a class metaobject is not built from scratch, but is created by passing an abstraction Δ and a (possibly empty) collection of parent-class metaobjects, written P^*, to a meta-class abstraction.

The reader should note that MetaModel and the meta-class entities are represented as *abstractions* (i.e. functions) whereas the class- and object-level entities are objects (i.e. record-like structures). Hence, unlike other meta-level approaches (e.g., Smalltalk or CLOS), class metaobjects are not instances of class metaobject classes.

4.1. *The* MetaModel *abstraction*

Using a similar notation as the one used for the classes Point and BoundedPoint, the abstraction MetaModel is defined as follows (note that we use $\lambda X \to b$ to denote an anonymous function with parameter X and function body b):

$$MetaModel(A) \quad = \quad \lambda X \to \mathit{fix}_{Mself} \, [\, \lambda Mself \to \mathit{Static\text{-}Protocol\text{-}Part} \oplus X \,]$$

$MetaModel$ is a function that expects a keyword-based argument A with bindings for G_m, W_m, and C_m, denoting the model generator, model wrapper, and model composer to instantiate a meta-class abstraction (e.g., Class) for a concrete semantic model. The result of $MetaModel$ is itself a function, which expects a keyword-based argument X with appropriate bindings for the concrete semantic model (e.g., Δ and P^* for Class). Evaluating this function yields a class-level object with self-reference $Mself$. This class-level object is defined as a polymorphic extension of the form

Static-Protocol-Part which defines the static protocol part of the meta-level framework with the keyword-based argument X passed to the meta-class abstraction. This application of polymorphic form extension enables overriding of the default generator, wrapper, and composer behaviour and is needed for modeling mixin application and composition. The form *Static-Protocol-Part* has the following structure:

$$
\begin{aligned}
Static\text{-}Protocol\text{-}Part \ = \ & \\
\{ \ G(I) \quad &\rightarrow \quad A.G_m(I \oplus X) \\
W(I) \quad &\rightarrow \quad \textit{fix}_s \, [\ A.W_m(\{init\!=\!I, \ self\!=\!s, \ G(Y) \rightarrow Mself.G(Y)\}) \ ] \\
C(M) \quad &\rightarrow \quad MetaModel(A)(\{G(I) \rightarrow A.C_m(\{mixin\!=\!M, \ init\!=\!I, \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad Mself\!=\!Mself\} \ ) \ \}) \ \}
\end{aligned}
$$

The default wrapper W passes (i) the init-parameter I, (ii) self, and (iii) the generator denoted by $Mself.G$ to the model wrapper W_m, and establishes the correct binding of self for the intermediate object created by W_m. In order to avoid interference of the three parameters and to keep things separate, we add additional structure to the parameter passed to W_m by using so-called *nested forms* [LUM 99]. Note that we use dynamic method binding in W for G in order to reflect the fact that the default generator may have been overridden by a composer C (see below).

The default generator G polymorphically extends the argument passed to itself with the bindings for the concrete semantic model and calls the model generator G_m with the resulting form.

The composer C provides the default behaviour for mixin application and composition. It creates a new class-level object based on the model abstractions passed to *MetaModel* and a composite generator defined by a model composer C_m. Again, nested forms are used to add additional structure to the argument passed to C_m. Note that mixin application and composition was not supported by our previous object model and, therefore, the classes Point and BoundedPoint defined in section 3 do not incorporate a composer abstraction.

4.2. *Class abstractions*

In this section, we will discuss model wrappers, generators, and composers for defining meta-class abstractions for class metaobjects. However, due to lack of space, we will only consider selected model abstractions which define single inheritance class models; refer to [SCH 99] for a discussion about other model abstractions, especially for multiple inheritance.

Common to all (single-inheritance) meta-class abstractions is that they require an abstraction Δ and a parent-class metaobject P^* as a parameter and that they share the same class model wrapper W_m^C and class model composer C_m^C. The two class model abstractions are defined as follows:

$$
W_m^C(I) \quad = \quad I.G \, (\{init\!=\!I.init, self\!=\!I.self\})
$$

$$C_m^C(M) \quad = \quad M.mixin.G \ (M.init \oplus \{P^* = M.Mself\})$$

The main difference between the meta-class abstractions for various class models lies in the way the model generators are defined:

$$G_m^S(I) \quad = \quad \text{let } Obj_I = I.P^*.G(I) \text{ in } Obj_I \oplus I.\Delta(I \oplus \{orig = Obj_I\})$$
$$G_m^B(I) \quad = \quad \text{let } Inner = I.P^*.\Delta_{Inner}() \oplus I.\Delta_{Inner}(), Obj_I = I.\Delta(Inner \oplus I) \text{ in }$$
$$Obj_I \oplus I.P^*.G(I \oplus Obj_I)$$

The model generator G_m^S defines a semantic model similar to Smalltalk (i.e. dynamic binding of **self**-calls, direct invocation of inherited methods etc.) whereas G_m^B defines a Beta model (i.e. a prefix-style of inheritance). Model generators for other schemes like static method dispatch can be defined similarly [SCH 99].

In fact, the model generator G_m^S is defined similarly to G_{BPoint}: it creates an intermediate object Obj_I of the parent-class (specified by $I.P^*$) and composes Obj_I with the intermediate object created by $I.\Delta$.

The model generator G_m^B for Beta requires an additional abstraction Δ_{Inner} (specifying a set of *null methods* for all methods defined in Δ [BRA 90]), which is passed to the corresponding meta-class abstraction together with Δ and P^*. Note that this Beta-specific extension of the meta-protocol substantially benefits from the keyword-based argument passing of the underlying FORM-calculus.

Using G_m^S, G_m^B, W_m^C, and C_m^C, the class abstractions Class specifying a Smalltalk-like class model and BetaClass specifying a Beta-style class model, respectively, are defined as follows:

$$Class(X) \quad = \quad MetaModel \ (\ \{G_m(I) \to G_m^S(I), W_m(I) \to W_m^C(I),$$
$$C_m(M) \to C_m^C(M)\} \ )(X)$$
$$BetaClass(X) \quad = \quad MetaModel \ (\ \{G_m(I) \to G_m^B(I), W_m(I) \to W_m^C(I),$$
$$C_m(M) \to C_m^C(M)\} \ )(X)$$

In order to break the recursion in the generators (e.g., the generator of a class metaobject always refers to the generator of its parent-class), it is necessary to introduce a common ancestor for all class-level objects which acts as the root of the class-level hierarchy and defines appropriate default behaviour.[1] In our framework, the root class is denoted by Object.

The classes Point and BoundedPoint can now be defined as follows:

$$Point \quad = \quad Class \ (\ \{P^* = Object, \Delta(I) \to \Delta_{Point}(I)\} \ )$$
$$BoundedPoint \quad = \quad Class \ (\ \{P^* = Point, \Delta(I) \to \Delta_{BPoint}(I)\} \ )$$

---

[1]Not all object-oriented programming languages have such a root class as a common ancestor to all classes (e.g., C++), but adding a root class does not change the underlying semantics and is considered to be a good programming practice.

4.3. *Mixins*

In order to overcome some of the problems with inheritance, several authors have proposed the notion of *mixins* [BRA 90, COO 89, VAN 96]. Mixins are usually intended to support some aspect of behaviour orthogonal to or independent of the behaviour supported by other classes. The main idea of this concept is that a mixin can be composed with (or mixed into) different parent-classes to create a related family of modified classes. Using the terminology of Van Limberghen and Mens, we will call such a composition *mixin application* [VAN 96]. Applying a mixin M to class A, written as $M * A$, results in a new class which combines the behaviour of both M and A with precedence of the behaviour of M.

As an example, reconsider the classes Point and BoundedPoint. The redefined method *move* in class BoundedPoint guarantees that the y coordinate of an instance never exceeds a given upper bound. The aspect of restricting the y coordinate in a method *move* is independent of the behaviour of the class Point and can, therefore, be factored out into a separate mixin, called Bound. Using the mixin Bound, the class BoundedPoint can now be defined by applying Bound to the class Point: BoundedPoint = Bound * Point.

The true value of mixins lies in that fact that they cannot only be applied to classes, but also composed with other mixins. Such an operation is called *mixin composition*. Mixin composition takes two mixins M_1 and M_2 and yields a composite mixin $M_1 * M_2$ which combines the behaviour of M_1 and M_2, but gives precedence to the behaviour of M_1. By definition, mixin application and composition are associative: $(M_1 * M_2) * A = M_1 * (M_2 * A)$ [BRA 90].

In our model, a mixin is an *abstract subclass* (i.e. a class definition with an unbound parent P^*). At the class level, mixins are represented as *mixin metaobjects* with an unbound parent P^*. As a consequence, a meta-class abstraction for mixins is solely an abstraction over Δ.

In order to handle mixin application and mixin composition uniformly, we define the concept of a *composer abstraction* C. A composer abstraction is a meta-level operation for both class and mixin metaobjects and takes as argument the left-hand side operand of a mixin application or composition. Hence, $M * A$ is modeled as $A.C(M)$ whereas $M_1 * M_2$ is encoded as $M_2.C(M_1)$.

Both mixin application and composition require a distinguished model composer C_m. In case of mixin application $M * A$, the model composer C_m^C is defined as shown in section 4.2. C_m^C returns an intermediate object by passing A as parent (denoted by P^*) to the generator of M.

In case of mixin composition $M_1 * M_2$, the situation is different. In the composite mixin, the parent of M_1 has to refer to M_2 whereas the parent of M_2 will be bound to a class, say A, by a final mixin application (e.g., $(M_1 * M_2) * A$). Therefore, the model composer C_m^M for mixin composition is defined as follows:

$$C_m^M(M) \quad = \quad \text{let } MetaObj = M.init.P^*.C(M.Mself) \text{ in}$$
$$M.mixin.G \ (M.init \oplus \{P^* = MetaObj\})$$

Within C_m^M, the subexpression $M.init.P^*.C(M.Mself)$ creates a new class metaobject $MetaObj$ which defines the application of the rightmost mixin to the class metaobject the composite mixin will be applied onto (e.g., applying $(M_1 * M_2)$ to a class metaobject A, then $MetaObj$ denotes $M_2 * A$). This class metaobject is used to bind the unbound parent in the leftmost mixin.

In general, defining a wrapper for a mixin M is useless since a mixin only specifies partial behaviour of objects. However, defining a mixin wrapper as an abstraction over a parent-class P^* allows us to define instances of *anonymous classes* $M * P^*$. Anonymous classes are useful if only a single instance is needed. This approach can be compared to anonymous lambda abstractions found in functional programming languages. A mixin model wrapper W_m^M is thus defined as follows:

$$W_m^M(I) \ = \ I.G \ (\ I \oplus \{P^* = I.init.P^*\} \ )$$

Using the the model generator G_m^S defined in section 4.2, the mixin model composer C_m^M, and the mixin model wrapper W_m^M, the meta-class abstraction Mixin (supporting Smalltalk semantics) is defined as:

$$Mixin(X) \quad = \quad MetaModel \ (\ \{G_m(I) \to G_m^S(I), W_m(I) \to W_m^M(I),$$
$$C_m(M) \to C_m^M(M)\} \ )(X)$$

The previously mentioned mixin Bound can now be defined as follows:

$$Bound \quad = \quad Mixin \ (\ \{\Delta(I) \to \Delta_{BPoint}(I)\} \ )$$

4.4. *Encapsulation*

In the literature, the term *encapsulation* has been employed with different meanings. In this work, we use the term encapsulation for *method hiding*. In this section, we illustrate the notion of an *encapsulation abstraction* (originally defined in [VAN 96]) and show that it can be most easily defined using our meta-level framework. Note that the encapsulation abstraction depends on polymorphic form restriction of the underlying FORM-calculus.

The encapsulation abstraction defined in our model can be seen as a variant of the *hide* operator presented by Bracha and Lindstrom [BRA 92]. It ensures that encapsulated methods become invisible to any client of a class-level entity encapsulation is applied to. Intuitively, encapsulating a method l of a class A consists of (i) replacing dynamically dispatched self-calls to l by statically dispatched calls and (ii) removing

l from the intermediate object created by the corresponding generator. Hence, the model generator G_m^E is defined as follows:

$$G_m^E(I) \quad = \quad \mathit{fix}_{\mathit{self'}}\, [\, \mathrm{let}\ \mathit{Obj}_I = I.P^*.G(I \oplus \{\mathit{self} = I.\mathit{self} \oplus \mathit{self'}\})\ \mathrm{in}$$
$$\mathit{Obj}_I \setminus I.\Delta(I \oplus \{\mathit{orig} = \mathit{Obj}_I, \mathit{self} = I.\mathit{self} \oplus \mathit{self'}\})\,]$$

In contrast to the class and mixin abstractions, the purpose of Δ is different as it does not specify an incremental behaviour, but the set of methods to be encapsulated.

Using the encapsulation model generator G_m^E, the mixin model composer C_m^M, and the mixin model wrapper W_m^M, the meta-class abstraction Encapsulate illustrated in Figure 2 is defined as follows:

$$Encapsulate(X) \quad = \quad MetaModel\,(\, \{G_m(I) \to G_m^E(I), W_m(I) \to W_m^M(I),$$
$$C_m(M) \to C_m^M(M)\}\,)(X)$$

As an example of how method encapsulation can be applied, consider the following specialization of the mixin Bound: LinearBound ensures the invariant that the y coordinate never exceeds the x coordinate (i.e. $y \leq x$):

$$LinearBound \quad = \quad Mixin\,(\{\Delta(I) \to \{bound() \to I.self.X()\}\}\,)\ *\ Bound$$

If we want to define a class DoubleBoundedPoint which adds the behaviour of both LinearBound and Bound to the class Point (i.e. the y coordinate never exceeds both the x coordinate and a given upper bound), it is not possible to define the class DoubleBoundedPoint as LinearBound $*$ Bound $*$ Point: the method $bound$ of LinearBound overrides $bound$ of Bound and, as a consequence, the test for the upper bound will not be performed. However, this problem can be solved by encapsulating the method $bound$ from Bound, written as Encaps (Bound, {bound}), and use the encapsulated Bound instead:

$$\mathsf{DoubleBoundedPoint} \ = \ \mathsf{LinearBound} * \mathsf{Encaps}\,(\mathsf{Bound}, \{\mathsf{bound}\}) * \mathsf{Point}$$

5. Conclusions and related work

We have presented a FORM-calculus based meta-level framework for modeling concurrent, object-oriented programming abstractions which incorporates the results of our previous work [LUM 96, SCH 97] and generalizes approaches presented by Bracha and Cook [BRA 90], Cook and Palsberg [COO 94], Van Limberghen and Mens [VAN 96], and Rossie et. al [ROS 96a]. By using a process calculus, concurrency does not need to be modeled explicitly, and the presented meta-level framework can also be used in a non-concurrent environment that provides the same expressive power for records as the FORM-calculus. In particular, keyword-based parameters as well as asymmetric record concatenation and restriction must be available.

The essentials of concurrent objects in the FORM-calculus are captured by our meta-level framework: an object is viewed as an agent containing a set of local agents

and channels representing methods and instance variables, respectively, whereas the interface of an object is a form containing bindings for the channels of all exported features. Furthermore, the framework defines an imperative object model with an early bing of self [BON 99], data encapsulation, and low-level synchronization [SCH 99].

Representing classes as first-class runtime entities (i.e. class metaobjects) allows us to integrate features of object-oriented programming such as class variables and methods, various inheritance mechanisms, and different method dispatch strategies. This is achieved by introducing intermediate objects (i.e. objects with an unbound self-reference) specifying partial behaviour of objects, generator agents defining compositions of intermediate objects, and wrappers, which apply a fixed-point operator over composed intermediate objects to establish a sound interpretation of self.

A generalization of the concepts of generators, wrappers, and composition of intermediate objects cannot only be used to define classes and class abstractions, but also to model mixins, mixin application, mixin composition as well as method encapsulation. By (i) splitting the functionality of generators and wrappers into a static protocol-part and a variable model-part and (ii) introducing the concept of a composer abstraction, it is possible to derive all object-oriented abstractions from a single meta-model abstraction. Whereas this meta-model abstraction defines the generic behaviour of all meta-class abstractions and the corresponding meta-protocol, so-called *model* generators, wrappers, and composers specify the common behaviour of specific semantic models. This approach also allows us to separate the semantic model of concrete classes and mixins from their specific behaviour.

In contrast to object models found in many object-oriented programming languages, our meta-level framework provides a compositional view of object-oriented programming abstractions: it makes a clear separation between functional elements (i.e. methods) and their compositions (i.e. inheritance). Furthermore, the meta-level framework enables the definition of various semantic models supporting different kinds of inheritance and method dispatch strategies, and clarifies concepts which are typically merged in existing programming languages. It is important to note that both keyword-based argument passing as well as polymorphic form extension and restriction are the key mechanisms to achieve the resulting flexibility and extensibility.

Several other models have been proposed for defining the semantics of concurrent object-based programming. Most of these models define objects and object-oriented abstractions as primitives, but they either hard-wire the underlying inheritance model [RED 88], integrate concepts in a non-orthogonal way [FLA 98, BON 99], or do not incorporate important features found in object-oriented programming languages (e.g. they lack inheritance [ABA 96]).

In contrast to object-oriented programming languages which represent classes as first-class entities (e.g., CLOS [KIC 91], Smalltalk, or Python [ROS 96b]), the framework presented in this work is much more general and defines less restrictions on the underlying model.

Acknowledgements

We thank all members of the Software Composition Group for their support of this work, especially Franz Achermann, Stéphane Ducasse, and Oscar Nierstrasz for helpful comments on an earlier draft.

6. References

[ABA 96] ABADI M., CARDELLI L., VISWANATHAN R., "An Interpretation of Objects and Object Types", *Proceedings POPL '96*, ACM Press, 1996, p. 396–409.

[AME 86] AMERICA P., DE BAKKER J., KOK J. N., RUTTEN J., "Operational Semantics of a Parallel Object-Oriented Language", *Proceedings of Principles of Programming Languages (POPL'86)*, 1986, p. 194–208.

[BAR 95] BARRIO SOLORZANO M., "Estudio de Aspectos Dinamicos en Sistemas Orientados al Objecto", PhD thesis, Universidad de Valladolid, 1995.

[BON 99] BONO V., PATEL A. J., SHMATIKOV V., "A Core Calculus of Classes and Mixins", GUERRAOUI R., Ed., *Proceedings ECOOP '99*, LNCS 1628, Springer, 1999, p. 43–66.

[BOU 92] BOUDOL G., "Asynchony and the π-calculus", Technical report num. 1702, INRIA Sophia-Antipolis, 1992.

[BRA 90] BRACHA G., COOK W., "Mixin-based Inheritance", MEYROWITZ N., Ed., *Proceedings OOPSLA/ECOOP '90*, vol. 25 of *ACM SIGPLAN Notices*, 1990, p. 303–311.

[BRA 92] BRACHA G., LINDSTROM G., "Modularity Meets Inheritance", *Proceedings of International Conference on Computer Languages*, IEEE Computer Society, 1992, p. 282–290.

[CAR 94] CARDELLI L., MITCHELL J. C., "Operations on Records", GUNTER C., MITCHELL J. C., Eds., *Theoretical Aspects of Object-Oriented Programming*, MIT Press, 1994.

[COO 89] COOK W. R., "A Denotational Semantics of Inheritance", PhD thesis, Department of Computer Science, Brown University, Providence, RI, 1989.

[COO 94] COOK W., PALSBERG J., "A Denotational Semantics of Inheritance and its Correctness", *Information and Computation*, vol. 114, num. 2, 1994, p. 329–350.

[FLA 98] FLATT M., KRISHNAMURTHI S., FELLEISEN M., "Classes and Mixins", *Proceedings POPL '98*, San Diego, 1998, ACM Press, p. 171–183.

[GOL 89] GOLDBERG A., ROBSON D., *Smalltalk-80: The Language*, Addison-Wesley, 1989.

[HON 92] HONDA K., TOKORO M., "On Asynchronous Communication Semantics", TOKORO M., NIERSTRASZ O., WEGNER P., Eds., *Object-Based Concurrent Computing*, LNCS 612, Springer, 1992, p. 21–52.

[KIC 91] KICZALES G., DES RIVIÈRES J., BOBROW D. G., *The Art of the Metaobject Protocol*, MIT Press, 1991.

[LUM 96] LUMPE M., SCHNEIDER J.-G., NIERSTRASZ O., "Using Metaobjects to Model Concurrent Objects with PICT", *Proceedings of Langages et Modèles à Objets '96*, Leysin, 1996, p. 1–12.

[LUM 99] LUMPE M., "A π-Calculus Based Approach to Software Composition", PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 1999.

[MAD 93] MADSEN O. L., MØLLER-PEDERSEN B., NYGAARD K., *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.

[MIL 90] MILNER R., "Functions as Processes",   *Proceedings ICALP '90*, LNCS 443, Springer, 1990, p. 167–180.

[MIL 91] MILNER R., "The Polyadic Pi-Calculus: a Tutorial", Technical report num. ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, UK, 1991.

[MIL 92] MILNER R., PARROW J., WALKER D., "A Calculus of Mobile Processes, Part I/II", *Information and Computation*, vol. 100, 1992, p. 1–77.

[PIE 95] PIERCE B. C., TURNER D. N., "Concurrent Objects in a Process Calculus", ITO T., YONEZAWA A., Eds., *Theory and Practice of Parallel Programming (TPPP)*, LNCS 907, Springer, 1995, p. 187–215.

[RED 88] REDDY U. S., "Objects as Closures: Abstract Semantics of Object Oriented Languages", *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, ACM, 1988, p. 289–297.

[ROS 96a] ROSSIE J. G., FRIEDMAN D. P., WAND M., "Modeling Subobject-based Inheritance", COINTE P., Ed., *Proceedings ECOOP '96*, LNCS 1098, Springer, 1996, p. 248–274.

[ROS 96b] VAN ROSSUM G., "Python Reference Manual", Technical report, Corporation for National Research Initiatives (CNRI), 1996.

[SAN 95] SANGIORGI D., "Lazy functions and mobile processes", Technical report num. RR-2515, INRIA Sophia-Antipolis, 1995.

[SCH 97] SCHNEIDER J.-G., LUMPE M., "Synchronizing Concurrent Objects in the Pi-Calculus", DUCOURNAU R., GARLATTI S., Eds., *Proceedings of Langages et Modèles à Objets '97*, Roscoff, 1997, Hermes, p. 61–76.

[SCH 99] SCHNEIDER J.-G., "Components, Scripts, and Glue: A conceptual framework for software composition", PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 1999.

[TUR 96] TURNER D. N., "The Polymorphic Pi-Calculus: Theory and Implementation", PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1996.

[VAN 96] VAN LIMBERGHEN M., MENS T., "Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems", *Object-Oriented Systems*, vol. 3, num. 1, 1996, p. 1–30, Chapmann & Hall.

[VAS 94] VASCONCELOS V. T., "Typed Concurrent Objects", TOKORO M., PARESCHI R., Eds., *Proceedings ECOOP '94*, LNCS 821, Springer, 1994, p. 100–117.

[WAL 95] WALKER D. J., "Objects in the Pi-Calculus", *Information and Computation*, vol. 116, num. 2, 1995, p. 253–271.