Vrije Universiteit Brussel
Faculteit Wetenschappen

A Formalisation of Encapsulated Modification of Objects

Kim Mens, Kris De Volder, Tom Mens

Techreport vub-prog-tr-96-06

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3525
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

A Formalisation of Encapsulated Modification of Objects

Kim Mens, Kris De Volder, Tom Mens

Department of Computer Science, Faculty of Sciences
Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium
e-mail: { kimmens | kdvolder | tommens }@vub.ac.be

Abstract. Currently existing formalisations of object orientation are unsatisfactory w.r.t. inheritance. This is due to an inherent conflict between inheritance and encapsulation. To solve this conflict, we present a formal model with a clear syntactic distinction between "inheritable entities" and objects. This model naturally gives rise to two types of incremental modification of objects without compromising encapsulation. Both types of modification are complementary from a software engineering point of view.

1 Introduction

1.1 Encapsulation versus Inheritance

Inheritance allows building new objects by incrementally modifying the implementation of existing objects. Encapsulation means that an object provides an abstraction barrier behind which implementation details can be hidden from the user. Since inheritance depends at least to some extent on implementation details, there is an inherent conflict with encapsulation. A number of problems in OO software engineering have arisen from this conflict, and the restriction of inheritance is under strong discussion at present in the OO community, as can be observed in the discussion about patterns, frameworks and separation of different kinds of interfaces.

In an attempt to solve this conflict, [Snyder 87] introduces a notion of *encapsulated inheritance*: inheriting clients have no direct access to the private attributes of their parents. However [Steyaert & De Meuter 95] show that this approach does not prevent all violations of encapsulation since late binding of self, which is crucial to inheritance, also involves a form of decapsulation. They argue that breaches of encapsulation by message passing clients can be inhibited if a clear distinction is made between inheriting clients and message passing clients. *Inheriting clients* access an object through its *specialisation interface* (cf. [Lamping 93]) exposing implementation details of the object important for inheritors, e.g. which method is called through self sends by what other method[1]. On the other hand, *message passing clients* can only access an object through its *client interface*, i.e. the set of all messages understood by the object.

There are two main models of inheritance employed in object oriented systems today: class based inheritance (e.g. SMALLTALK [Goldberg & Robson 83] and C++ [Stroustrup 92]) and prototype based inheritance (e.g. SELF [Ungar & Smith 87]). Class based systems provide classes as explicit entities for structuring inheritance hierarchies. In these systems, inheritance is restricted to classes and is not defined on objects directly. This restriction enables the successful integration of inheritance and encapsulation but prohibits dynamic object extension. On the contrary, in prototype based languages no distinction is made between objects and "inheritable entities". Instead objects are modified directly. This is particularly suitable in situations where the

[1] This information is important for inheritors since it provides knowledge about the effect of overriding a method.

dynamic evolution of objects is desired. However this is achieved at the expense of an object model which does not maintain the encapsulation property, as discussed by [Steyaert & De Meuter 95], [Mezini 95] and [Snyder 87].

1.2 Encapsulated Modification of Objects

According to [Dony et al. 92] "languages without dynamic modification of structures enforce a robust and more disciplined view of prototype based programming while other languages provide some flexibility which can hurt more than it may help". A drastic solution to this problem is to forbid all dynamic modifications (as in the class based approach). However we prefer to find a prototype based model which offers dynamic extension of objects without compromising encapsulation.

Research at our lab has been very concerned with finding such a model. This ongoing effort has lead to the design of a prototype based language called AGORA ([Steyaert et al. 93], [Codenie et al. 94]) which provides mixin methods as an encapsulated form of the pure mixins proposed by [Bracha & Cook 90]. In [Steyaert & De Meuter 95] it is argumented that mixin methods are a form of *encapsulated inheritance*[2] (i.e. they do not compromise encapsulation) by indicating that it conforms to the "immaculate client interface principle for objects":

> *An object should not expose any other interface but the client interface to its message passing clients. It must be able to hide its specialisation interface from message passing clients.*

This is a sufficient condition to ensure that an object's message passing clients cannot breach encapsulation. Taking into account all arguments stated in this section, we demand that the prototype based model we are looking for has the following characteristics:

(a) It provides a means for incremental modification;
(b) Such modifications are done dynamically on objects;
(c) The model satisfies the immaculate client interface principle.
A model or language satisfying these demands has *encapsulated modification of objects*.

1.3 Overview of the Paper

In this paper we propose a formal model that captures the essence of encapsulated modification of objects. As in class based systems, our model distinguishes inheritable entities from objects. However, it is more flexible because objects can explicitly manipulate the inheritable entities albeit in a restricted way.

We will show how the model naturally gives rise to two types of incremental modification, namely *(encapsulated) inheritance* and *conservative modification*. These are complementary from a software engineering point of view because they make use of an object's specialisation interface and client interface respectively. Inheritance should be restricted because it depends on the implementation details exposed in the specialisation interface and therefore threatens encapsulation. Conservative modifications can be applied unrestrictedly because they only depend on the client interface.

In section 2 we start out with describing the criteria that guided us in designing the formal model and present the formal syntax and semantics. In section 3 we show how the model allows us to define the two types of

[2] Not to be confused with the notion of encapsulated inheritance as introduced by [Snyder 87].

incremental modification on objects without endangering encapsulation. Section 4 discusses how the restrictions our model imposes are beneficial for software development. In section 5 we explain the drawbacks and show that they are not inherently connected with the concept of encapsulated modification of objects but rather due to the simplicity of the model. Section 6 discusses some related work. Section 7 indicates some research topics that are worth investigating in the future and section 8 summarises the conclusions of the paper.

2 The Formal Model

2.1 Design Guidelines

Because of the proliferation of object orientation in recent years, it is very difficult to give a general characterisation. Nevertheless, we will identify the concepts and principles we deem essential to the different object paradigms. This assessment will be used as a general guideline in the design of our formal model.

A rigorous object oriented model should be uniform: any (first-class) entity is an object and the unique control structure is message passing. Furthermore, as in [Steyaert 94] we demand that message passing is atomic. This means that a message send cannot be decomposed into more elementary operations. Of course we also require our model to have encapsulated modification of objects, to build new objects out of old ones. Finally, we deem late binding of self to be essential since typical object oriented programming techniques make heavy use of it and gain much of their power from it. Summarising all of this we state the following design principles:

(1)	Objects (and objects only) are first-class citizens.
(2)	Atomic message passing is the only control structure.
(3)	The model has encapsulated modification on objects.
(4)	An incremental modification mechanism with late binding of self is needed.

2.2 A Layered Syntax

The main idea behind the model is to syntactically distinguish first-class objects from inheritable entities. The inheritable entities are called *generators* in accordance to [Cook 89]. This yields a two-layered syntax. One layer for objects and message passing and a second (sub-)layer for generators and explicit generator manipulation. This does not contradict design guideline (1) because the entities of the second layer are not first-class citizens in the model.

As originally stated in [Cook 89] an object is simply an environment of methods, while a generator is a template for an object in the sense that its “self” remains to be filled in. Since the self of a generator can be filled in at will, unrestricted manipulation of explicit generators at language level breaches encapsulation (see section 1.1). This is the case in existing prototype based models and languages which do not distinguish objects from generators. In our model, the use of generators can be appropriately restricted because the layering offers a rigid syntactic distinction between objects and generators.

2.3 Formal Syntax

In this section we present the formal grammar describing the syntax of the model. In order to highlight the essence of encapsulated inheritance on objects other features such as typing, object identity, state and private attributes are not included in the model.

The start symbol is `Object` and terminal symbols are printed in bold. An identifier `Ident` is also considered to be a terminal symbol. The set of all object expressions is defined by the following productions.

Object	→	Object.Ident(Object)	*message send*
	|	[Generator]	*object creation*
	|	Ident	*argument reference*

This definition declares message sending as an operation on objects. Sending a message `m` with argument object `a` to a receiver object `o` is denoted `o.m(a)` and yields a new object. This is an atomic operation since it cannot be decomposed into more primitive operations.

In the definition above, identifiers[3] are used to represent (argument) objects. However, they can also be used to represent method names or generators, as can be seen in the production rules for generator expressions:

Generator	→	Generator ; Generator	*composition*
	|	Ident(Ident) = Ident#Object	*method*
	|	> Object <	*object generator*
	|	Ident	*self reference*
	|	ε	*empty generator*

The use of generators is severely restricted. They cannot be passed around as arguments, nor be returned as values from a message send, nor be sent messages. They can only be composed with each other. We will see in section 3.1 that this composition can be used to model an incremental modification mechanism with late binding of self, thus satisfying guideline (3).

Since an object is an environment of methods and is created by means of a generator, the most primitive kind of generator (apart from the empty generator) is a single method. The body of a method can refer to the late bound *self* of its object to perform self sends or to extend the object. Inside the body of a method such a self reference denotes the generator from which the inner-most object enclosing the method is created. Self references can be named by means of the `#` binding-operator[4]. Naming of self references is not only convenient but effectively increases the power of the model, because it allows multiple self references with different names at different nesting levels to be accessed from within the same method body.

As an example of an object created by a composition of methods, consider the following representation of a person named Joe[5]. It contains a method `isThatYou` performing a self send of the `name` message:

```
[       name(dummy) = Self # "Joe";
        isThatYou(who) = Self # [Self].name(dummy).equal(who)
]
```

Intuitively, `m(a)=S#`*`body`* means that the name `S` is going to be used to denote self references inside the body of `m`. For example, the identifier `Self` used inside the body of `isThatYou` refers to the following generator:

```
name(dummy) = Self # "Joe";
isThatYou(who) = Self # [Self].name(dummy).equal(who)
```

The above example also illustrates the use of arguments. The `isThatYou` method expects an argument object that can be referred to by `who` inside the body of the method. At message sending time, the formal argument `who` will be replaced by the actual argument supplied in the method call. For example, sending the message

---

[3] In all our examples we will take the convention that identifiers denoting objects start with a lower-case letter, whereas identifiers denoting generators start with an upper-case letter.

[4] Neglecting syntactic differences, this binding operator corresponds to the σ() operator as defined in [Abadi & Cardelli 94].

[5] Although not explicitly present in the syntax, throughout the examples we will make use of predefined strings understanding an `add` message to concatenate them with other strings, and an `equal` method for equality testing.

`isThatYou("Joe")` to the object returns the `TRUE` object as a result.

Notice that the syntax allows only one argument to be passed with a message. This is not a restriction since multiple arguments can be simulated in the form of an object where the arguments are listed as methods.

In what follows, we assume the following conventions:

- When no actual argument is supplied in a message send, the formal argument is substituted with the empty object `[ε]`.
- Vice versa, one does not need to write a formal argument that is not referred to in the body of a method. In this case an actual argument supplied in a message send is simply ignored.
- When no reference to the self generator is made inside the body of a method, we agree to omit the local reference name.

With all these conventions, the example becomes:

```
[       name = "Joe";
        isThatYou(who) = Self # [Self].name.equal(who)
]
```

Finally, the **>…<** operator will allow the extension of an object *without* late binding. It turns an object into a generator with fixed self. The usage and implications of this generator will become clear in section 3.2.

2.4 Examples

To illustrate the use of "self references" we show how two mutually recursive objects – the Boolean values `true` and `false` – can be implemented by embedding them in the same environment (referred to by `Env`), and by making "self calls" to this environment from within the objects. Notice how the same environment `Env` can also be used to test the example.

```
[       true = Env # [        if(action) = action.then;
                              not = [Env].false;
                              or(boolean) = Self # [Self];
                              and(boolean) = boolean              ];
        false = Env # [       if(action) = action.else;
                              not = [Env].true;
                              or(boolean) = boolean;
                              and(boolean) = Self # [Self]        ];
        test = Env # [Env].false.not.or([Env].true)
                              .if([  then=[Env].true; else=[Env].false ])
].test
```

It is important to note that self references refer to the generator from which the self object is created rather than to the object itself. This enables extending objects in a late bound way by adding more methods to (or overriding old methods of) the *self* generator. For example, consider the following simulation of an object with state:

```
[       getName = "Jones";
        setName(newName) = Self # [ Self ; getName = newName ]    ]
```

Sending the message `setName("Smith")` to this object adds a method `getName = "Smith"` to the generator. Because this method is added to the right of the already present methods, and because the semantics will ne defined in such a way that method-lookup occurs from right to left, this has the effect of overriding the old `getName` method.

2.5 The Δ-operator

Since generators are templates for objects with a still undetermined self, it is natural to define an operator that

explicitly binds the self of a generator. We use the symbol Δ to denote this operator. The self of a generator is again a generator in our model, thus the Δ-operator is a binary operator on generators returning an object. Basically, the Δ-operator is a delegation operator. Its first argument is a generator in which messages sent to the resulting object will be looked up (thus constituting a kind of client interface). The second argument represents the generator that is used for internal self references such as self sends (thus constituting a kind of specialisation interface).

In order to introduce the Δ-operator, the object syntax described in section 2.3 needs to be extended with one new kind of object expression:

```
Object    →  …
          |  [ Generator Δ Generator ]                    object creation
```

An expression of this form is also called an *object creation*, because from now on we consider the form `[G]` to be syntactic sugar for `[GΔG]`[6]. In section 3.3 an example will be given of how this operator can be used to perform super calls.

2.6 Formal Semantics

The denotational semantics of our formal model follows the notation of [Schmidt 86]. Due to space limitations, the operational semantics and some interesting theoretical properties thereof, such as confluence and a translation of λ-calculus into our model are left out from the paper.

Syntactic Domains

```
ObjExpr      =    set of all syntactic object expressions
GenExpr      =    set of all syntactic generator expressions
Ident        =    set of all syntactic identifiers
```

Semantic Domains

As in most denotational semantics of object oriented models presented in literature ([Cook 89], [Cardelli & Mitchell 89], [Wegner & Zdonik 88]), an `Object` is merely an environment of methods. A `Method` expects an `Object` as argument and returns an `Object` after evaluation. For reasons described in [Steyaert & De Meuter 95], a `Generator` is a function mapping a `Generator` onto an `Object`. Square brackets are used for parameterised domains.

```
Env[α]       =    Ident → Maybe[α]
Maybe[α]     =    α ⊕ Unit
Object       =    Env[Method]
Method       =    Object → Object
Generator    =    Generator → Object
```

Auxiliary Functions

Before presenting the semantic functions we need some auxiliary functions to create and manipulate environments.

```
{}           :    Env[α]                                  An empty environment
{}           =    λx.inMaybe[α]()
```

[6] To avoid confusion we point out that […Δ…] is an atomic operation, taking two generators and returning an object. It is *not* the compostion of …Δ… and […]. Without the enclosing […], …Δ… is not a valid expression, it is neither an object nor a generator expression.

```
{…→…}          :    Ident → α → Env[α]                    A single slot environment
{key→val}      =    λx.(x=key → inMaybe[α](val) , inMaybe[α]())

… +r …         :    Env[α] → Env[α] → Env[α]              Right preferential concatenation
f1 +r f2       =    λx.cases (f2 x) of
                         isUnit() → f1 x ,
                         isα()    → f2 x
                    end

lookup         :    Env[α] → Ident → α
lookup         =    λr.λk.cases (r k) of
                           isα(v)   → v ,
                           isUnit() → ⊥
                      end
```

Semantics of an Object Expression

The semantics of an object expression **ObjExpr** is a function expecting an environment of objects (the list of formal parameters bound to the actual arguments) and an environment of generators (referring to the selves of objects in which the current method is nested), and returning an object.

$[\![\textbf{ObjExpr}]\!]_O$: Env[Object] → Env[Generator] → Object

The semantics of a message send O_r.I(O_a) is defined by looking up the message I in the receiver object O_r, while providing the object O_a as argument.

$[\![O_r.I(O_a)]\!]_O$ = λe.λc.lookup ($[\![O_r]\!]_O$ e c) I ($[\![O_a]\!]_O$ e c)

The semantics of an object creation $[G_1 \Delta G_2]$ creates an object from G_1 by installing G_2 as its self.

$[\![\ [G_1 \Delta G_2]\ ]\!]_O$ = λe.λc.($[\![G_1]\!]_G$ e c) ($[\![G_2]\!]_G$ e c)

The object creation expression [G] is a convenient abbreviation for [GΔG] which creates an object from a generator by making the generator refer to itself.

$[\![[G]]\!]_O$ = $[\![[G \Delta G]]\!]_O$
= λe.λc.($[\![G]\!]_G$ e c) ($[\![G]\!]_G$ e c)

Evaluation of an identifier on object level occurs by looking up this identifier in the environment of actual arguments.

$[\![I]\!]_O$ = λe.λc.lookup e I

Semantics of a Generator Expression

The semantics of a generator expression **GenExpr** is a function expecting an environment of objects (the list of bindings of formal arguments to actual values) and an environment of generators (referring to the selves of objects in which the current generator is nested), and it returns a Generator as result.

$[\![\textbf{GenExpr}]\!]_G$: Env[Object] → Env[Generator] → Generator

Composition of two generators corresponds to right preferential concatenation of their method environments.

$[\![G_1;G_2]\!]_G$ = λe.λc.λs.($[\![G_1]\!]_G$ e c) s +r ($[\![G_2]\!]_G$ e c) s

The semantics of a method generator is a single slot environment that binds the method name to its body. This body is a function expecting an argument, and returning an object after having evaluated the body in the environment of local argument bindings extended with the argument passed to the method, and in the environment of generators extended with the current self generator.

$$[\![I_m(I_a)=I_s\#O]\!]_G = \lambda e.\lambda c.\lambda s.\{I_m \rightarrow body\}$$
$$\textit{where } body = \lambda a.[\![O]\!]_O\ (e +_r \{I_a \rightarrow a\})\ (c +_r \{I_s \rightarrow s\})$$

The semantics of an object considered as generator is a generator which ignores its self parameter and simply returns the object. Extending such a generator will therefore have no influence on the self of the object, i.e. the object's self is not subject to late binding. The use of such generators will be discussed further in section 3.2.

$$[\![>O<]\!]_G = \lambda e.\lambda c.\lambda s.[\![O]\!]_O\ e\ c$$

Evaluation of an identifier on generator level looks up this identifier in the environment of self generators. Finally, an empty generator returns an empty object which is just an empty environment.

$$[\![I]\!]_G = \lambda e.\lambda c.\mathtt{lookup\ c\ I}$$
$$[\![\varepsilon]\!]_G = \lambda e.\lambda c.\lambda s.\{\}$$

3 Encapsulated Modification of Objects

Our model has encapsulated modification of objects (as defined in section 1.2).

- First of all, as will be discussed in detail in the rest of this section it provides a means for incremental modifications.
- Secondly these modifications are applied dynamically on objects.
- Finally, from the denotational semantics we conclude that the immaculate client interface principle is satisfied. This is evident in the object model: an object is merely an environment of methods. Therefore on the semantic level an object is fully characterised by how it reacts to messages and the only way to interact with an object is through its client interface (i.e. the set of all messages understood by the object).

3.1 Two types of Incremental Modification

The model supports two types of incremental modification. On the one hand a mechanism with late binding of self, but restricted to a set of privileged inheriting clients which can access the specialisation interface. On the other hand a mechanism without late binding, by unprivileged clients using only the client interface of the object. The former mechanism will be referred to as *inheritance*, while the term *conservative modification* will be reserved for the latter. Inheritance tends to be implementation dependent, whereas conservative modifications are of an abstract nature since they can only access the client interface.

3.2 Conservative Modification

Unprivileged clients that only have access to the client interface can modify an object by using the **>…<** operator which casts an object into a generator that can be extended afterwards. We have seen in the semantics that such generators ignore their self argument. This means that late binding of self does not apply. Therefore such modifications are called *conservative* since they embed the object as is, without changing its internal workings. Consider for example[7] the following object for making points in Cartesian coordinates:

[7] In this example we use numerals as basic primitives understanding an `add`, `sqr`, `abs` and `sqrt` message. We will show in section 5 how numerals can be implemented in our model.

```
[ makeCartesian(arglist) =
    [ getx = arglist.x;
      gety = arglist.y;
      distance = Self # [Self].sumOfSquares.sqrt;
      sumOfSquares = Self # [Self].getx.sqr.add([Self].gety.sqr) ];
  cartesianPoint = Env # [Env].makeCartesian([x=3;y=4])
]
```

We can override the `distance` method of `cartesianPoint` with a method that computes the Manhattan rather than the Euclidean distance.

```
[ makeCartesian(arglist) = … ;
  cartesianPoint = Env # [Env].makeCartesian([x=3;y=4]);
  manhattanPoint = Env #
    [ >[Env].cartesianPoint<;
     distance = Self # [Self].getx.abs.add([Self].gety.abs) ]
]
```

This modification is conservative because it only depends on the abstract functionality offered by the `cartesianPoint` object's client interface. The inheritor does not need to know which self sends are performed in the object.

3.3 Inheritance

Inheritance is accomplished by adding methods directly to an object's self generator. One could call this "extension from the inside" of an object because the visibility of the generator representing the self of an object is restricted to code nested inside the object. Since generators are not first class and cannot be passed around as arguments nor be returned as values, this scoping mechanism limits the use of the self generator (and thus the use of inheritance) to the set of objects and methods whose declarations are nested inside the object the self belongs to.

As a first example of inheritance, consider an object representing a person with public attributes `name`, `sex` and `title`.

```
[ name    = "Ann Onymous";
  sex     = "Female";
  title   = "Miss ";
  letterHead = Self # [Self].title.add([Self].name)
]
```

When the message `letterHead` is sent to the object, the name is returned with the correct title prefixed to it. Now suppose that we want to use the above object as a prototype for producing new objects with a similar behaviour. For this purpose it needs to be extended with a new method `newPerson` modifying the original object. This method expects two arguments representing the name and sex of the new person. It is not necessary to pass the title of the new person as an argument since this can be determined by inspecting the sex.

```
[ name = …; …; letterhead = …;
  newPerson(init) = Self #
       [ Self; name = init.name; sex = init.sex;
         title = init.sex.equal("Female").if([then="Miss ";else="Mr. "]) ]
]
```

Next consider a more general implementation of the person object that anticipates the overriding of the `sex` attribute, by using a self send in its implementation of the `title` method. This facilitates the implementation of the `newPerson` modification.

```
[ name = "Ann Ticipate";
  sex = "Female";
  title   = Self # [Self].sex.equal("Female").if([then="Miss ";else="Mr. "]);
  letterHead = Self # [Self].title.add([Self].name);
  newPerson(init) = Self # [ Self; name = init.name; sex = init.sex ]
]
```

The `newPerson` method uses inheritance to override the attributes `name` and `sex`. This is true inheritance and cannot be accomplished by conservative modification because it makes use of implementation details of the person object. The implementation of `newPerson` clearly depends on whether or not `title` anticipates the overriding of `sex` and performs a self send to it.

As another example of inheritance suppose we want to allow a person to get married. This implies changing the title of a female person to `"Mrs."`, so the `title` method's implementation needs to be overwritten. In the example below, sending `getMarried` appropriately overrides the `title` method.

```
[ name = …; … ; newPerson(init) = …;
  getMarried = Self # [Self;
        title  = Self # [Self].sex.equal("Female").if([then="Mrs. ";else="Mr. "])]
]
```

In the AGORA language ([Steyaert et al. 93], [Codenie et al. 94]), this kind of incremental modification is referred to as *mixin method based inheritance* and is the only way to incrementally modify an object. In AGORA terminology, the `getMarried` method is called a *mixin method* since it contains additional behaviour that should be "mixed in" the object.

Using the Δ-operator, it is possible to model super sends to invoke parent operations in an inheritance chain. Consider the example of a two dimensional point which can be extended to a three dimensional point. The three dimensional point is created by overriding the parent's `sumOfSquares` method and uses a super call to implement the new `sumOfSquares` method.

```
[ x = 1; setx(newx) = Self # [Self ; x = newx];
  y = 2; sety(newy) = Self # [Self ; y = newy];
  isOrigin = Self # [Self].distance.isZero;
  distance = Self # [Self].sumOfSquares.sqrt;
  sumOfSquares = Self # [Self].getx.sqr.add([Self].gety.sqr);
  thirdDimension = Super # [
    Super;
    z = 3; setz(newz) = Self # [Self ; z = newz];
    sumOfSquares = Self # [SuperΔSelf].sumOfSquares.add([Self].getz.sqr) ]
]
```

Super calls are looked up in the parent's generator (`Super`), while the internal self references of the parent are redirected to the specialisation (`Self`), so messages to `[SuperΔSelf]` behave like super sends as in SMALLTALK. It is also possible to invoke operations in non-direct parents by means of the explicit naming of self generators. This can be helpful in programming tasks requiring multiple inheritance. Mixin methods constitute an inheritance mechanism that explicitly linearises the inheritance chain. This is a problem when dealing with multiple inheritance. When linearising a multiple inheritance graph, we need a mechanism to refer to non-direct superclasses. In [Boyen et al. 94] such a mechanism –called *stubs*– is introduced for AGORA. By inserting stubs at the right place in the inheritance chain, inheritors can use non-direct parents as parameters and "mimic" a graph structure in the linear chain. Stubs serve as pointers to the place in the inheritance chain where method lookup should start when invoking parent operations. We can easily model stubs, but because of space limitations we won't explain this any further.

4 Virtues of Encapsulated Modification of Objects

As already mentioned in section 1.1, most current day prototype based languages offer no such thing as encapsulated modification of objects, due to their lack of encapsulation. In this section we will illustrate that languages with encapsulation on objects are preferable for software engineering purposes, since they enforce more safety, facilitating reuse.

4.1 Encapsulation Enforces more Safety

Encapsulated modification of objects provides an incremental modification mechanism which restricts access to the specialisation interface to a well determined set of privileged inheriting clients. This enforces more safety, as illustrated by the following example.

Consider a bank account as a stand-alone entity that is not subject to late binding. It is however possible to modify the object from the inside. In this way, the account object can be extended before using it, for example to make it a safe bank account sealed with a password.

An `account` object created with the `makeAccount` method contains the `amount` of money present, the `name` of the account owner, and methods to `withdraw` money from or `deposit` money on the account. By invoking `secureAccount` the account can be modified from the inside to make it a safe account, so that money can only be withdrawn if a correct password is provided. Each time a `withdraw` is requested, a self send to `checkValidity` is performed to test if the correct password was provided.

```
[ true = …; false = …;
  makeAccount(owner) =
    [ name = owner;
      amount = 0;
      withdraw(value) = Self #
         [ Self; amount=[Self].amount.subtract(value) ];
      deposit(value) = Self #
         [ Self; amount=[Self].amount.add(value) ];
      secureAccount(secret) = Super #
        [ Super;
           secureAccount = "Warning: this method can only be called once!";
           checkValidity(pwd) = Self # secret.equals(pwd);
           withdraw(arg) = Self # [Self].checkValidity(arg.password)
                  .if([ then = [SuperΔSelf].withdraw(arg.value);
                        else = "Warning: your password is incorrect!" ])
        ]
    ];
  account = Env #
     [Env].makeAccount("Jones").secureAccount("gf&452aQ")
          .deposit(1000).withdraw([value=500;password="gf&452aQ"])
          .setPassword([old="gf&452aQ";new="%6*tyusQ"]);
  stealMoney(amount) = Env #
     [ >[Env].account<;
       checkValidity(pwd) = [Env].true
     ].withdraw([value=amount;password="Don't care"])
].stealMoney(500)
```

To illustrate the safety of the above defined `account` object we try to steal money by withdrawing money from the account without providing the correct password. Our attempt consists of overriding the `checkValidity` method with a new one that always returns `true`. Since self sends inside `account` are not bound late (`account` is fully encapsulated), all internal self sends to `checkValidity` keep referring to the original version, so cheating the `account` object in such a way is impossible. This is exactly the strength of encapsulated modification of objects. The user of an object cannot inadvertently misuse an object if the programmer does not allow this.

Note that it is also impossible to override the password of the secured account by trying to install a new password directly through the `secureAccount` message. Indeed, after the `secureAccount` method has been invoked once it is cancelled immediately by overriding it with a new method which returns a warning.

4.2 Substitutability of Objects

For software engineering purposes it is vital that whenever two objects have the same or similar behaviour from message passing point of view, they should be substitutable for one another, i.e. code that works correctly with one object should also work properly with the other one. Although this may seem rather trivial, it certainly is not the case in current prototype based languages where an object exposes its specialisation

interface.

Substitutability is intimately related with encapsulation of implementation details. In order to be able to substitute an object by another one with the same behaviour but a different implementation, clients using the object should not be aware of its implementation details. Consider the following simple example of two objects `cartesianPoint` and `polarPoint` which have exactly the same behaviour when observed from a message passing client's point of view.

```
[ makeCartesian(arglist)= …as in section 3.2… ;
  makePolar(arglist)=
     [ getx=arglist.r.times(arglist.teta.cos);
       gety=arglist.r.times(arglist.teta.sin);
       distance=arglist.r;
       sumOfSquares=Self#[Self].distance.sqr ];
  cartesianPoint=Env#[Env].makeCartesian([x=0;y=1]);
  polarPoint=Env#[Env].makePolar([r=1;teta=π/2])
]
```

While `cartesianPoint` and `polarPoint` are substitutable for each other in our model, this would cause errors in a model without encapsulated modification. The problem is that the point objects have different specialisation interfaces: in `cartesianPoint`, `distance` performs a self send to `sumOfSquares`, while in `polarPoint` `sumOfSquares` performs a self send to `distance`. Thus overriding the `distance` method will have a different effect on each point.

In presence of non-encapsulated modification, overriding a method always involves late binding. Therefore objects with different specialisation interfaces are not substitutable in such models. As an example reconsider overriding the `distance` method so that it implements the Manhattan distance. This works perfectly in `cartesianPoint`, but not in `polarPoint` since the `sumOfSquares` method in `polarPoint` is implemented with a self send to `distance` so it would yield an incorrect result.

With encapsulated modification self sends are not visible to unprivileged clients. In our model one can safely override the `distance` method of any version of `point` using a conservative modification:

```
[ point = …a polar point or a Cartesian point…;
  manhattanPoint = Env #
     [ >[Env].point<;
       distance = Self # [Self].getx.abs.add([Self].gety.abs) ]
]
```

This works correctly regardless of the implementation of `point` because self sends to the `distance` method in the original point will not be bound late and are not affected by overriding the method.

5 Shortcomings of the Model

Ideally all modifications which only require an object's client interface must be possible in a conservative way. We have shown an example (Cartesian and polar points) where this is feasible, however there are cases where conservative modifications are not as elegant as one would want. They all have in common that they involve methods which return self references in some way.

As an example consider the following implementation of positive integers. The only thing a positive integer understands are the messages `ifZero`, `pred` and `succ`. The `succ` method in the `zero` numeral[8] is defined as a "mixin method" returning the next numeral by inheriting the code of `zero` and overriding the `ifZero` and `pred`

[8] The `zero` numeral does not understand a `pred` message since we only consider positive numerals.

method.

```
[ zero = [ ifZero(action) = action.then;
           succ = Pred #
             [ Pred;
               ifZero(action) = action.else;
               pred = [Pred]
             ]
         ]
]
```

Using only `ifZero`, `pred` and `succ`, all operations on positive integers can be abstractly defined. Hence it should be possible to add these operations from the outside using only the client interface. For example we can extend integers with an `add` method as shown below.

```
[ zero = …as defined above…;
  withAdd(number) = Env #
    [ >number<;
      pred = [Env].withAdd(number.pred);
      succ = [Env].withAdd(number.succ);
      add(n) = Self # [Self].ifZero([ then = n;
                                      else = [Self].pred.add(n.succ) ])
    ];
  zeroWithAdd = Env # [Env].withAdd([Env].zero);
  one    = Env # [Env].zeroWithAdd.succ;
  two    = Env # [Env].one.succ;
  four   = Env # [Env].two.add([Env].two);
  test   = Env # [Env].four.pred.pred.pred.pred.ifZero([then=1;else=0])
]
```

This solution is not entirely satisfactory because it is necessary to "patch" the `succ` and `pred` messages. Otherwise `add` would only be understood by the `zeroWithAdd` numeral since conservative modifications do not affect self references in the object (no late binding). Both the results of `succ` and `pred` are constructed through self references. In the case of `succ` the successor is an extension of the self. In the case of `pred` it is the self of the parent object.

In general it is necessary to patch any method which directly or indirectly returns a self reference. This would not be problematic if it were not the case that one often wants to apply the same extension to different objects. Each different object implies patching up a potentially different set of messages. This problem already manifests itself in the example: the `pred` method is always patched even for a zero numeral which does not possess a `pred` method.

In a real programming environment one would like the patches to be applied automatically where needed, thus alleviating the problem. We have implemented an elegant solution that infers the needed information from a method's implementation. This solution enables a more sophisticated **>…<** operator to dynamically detect whether a method returns a self reference (or a derivation thereof) and automatically patch the returned result. Since the semantics is rather difficult we have chosen not to present it in this paper.

6 Related Work

An approach that is very similar to one described here is the delegation based object calculus developed in [Fisher & Mitchell 95]. In that paper, an object calculus is presented by adding new syntactic forms on top of untyped lambda calculus. A problem with this approach is that functions are first class, while we prefer a pure object model, containing only objects as first-class entities. By adding a type system to their basic calculus, Fisher and Mitchell make an explicit difference between objects and inheritable entities called *prototypes*. This approach differs from ours, from a practical as well as a theoretical point of view:

- The theoretical difference is that we have embedded the distinction between objects and inheritable entities in the basic syntax and semantics of the calculus, so we don't need an additional type system for this purpose.

- From a pragmatic point of view, the difference lies in the motivation of the calculus: Fisher and Mitchell developed a new type system in order to solve problems in existing type systems (such as [Abadi & Cardelli 94]), while we have developed a two-layered calculus in order to cope with the encapsulation problems in current object oriented languages (cf. [Snyder 87], [Steyaert & De Meuter 95]).

When comparing both calculi more closely, one can found two other differences:

- A minor difference between both approaches lies in the functionality of objects and inheritable entities. In [Fisher & Mitchell 95] the functionality of objects forms a subset of the possible operations on prototypes: while prototypes can be used to send messages and to add or redefine methods, an object can only be sent messages to. We prefer to keep the functionality of both kinds of entities separated for reasons of orthogonality: an object can only be sent messages to, while a generator can only be used to add or redefine methods.
- A more essential difference lies in the epressivity of objects. While in the Fisher/Mitchell calculus methods in an object are allowed to to redefine themselves or other methods, they are not allowed to add new methods, since this would give rise to type problems. Nevertheless, the latter kind of behaviour is really desired, as shown in section 3.3. It corresponds to the concept of mixin-method based inheritance in the Agora language.

In [Abadi & Cardelli 94] a simple object calculus supporting method override and object subsumption is introduced. Subsumption is the ability to emulate an object by means of another object that has more refined methods. Override is the operation that modifies the behaviour of an object by replacing one of its methods. As in our approach, instead of struggling with complex encodings of objects as λ-terms or other primitive constructs, objects are taken as primitives themselves. The object calculus however does not have encapsulated modification of objects. An example similar to our “bank account” example of section 4.1 can be used to show that local variables can be exported using a clever overriding of methods. Moreover the mechanisms for incremental modification in this calculus are too restrictive. Only overriding of already defined methods is allowed. It is impossible to add new methods to an already existing object.

Some of the ideas of the current model are derived from the OPUS-calculus presented in [Mens et al. 94]. This calculus satisfies encapsulated inheritance in the sense of [Snyder 87], and with a very slight modification even encapsulated modification on objects. Moreover it contains private attributes as basic primitives. Nevertheless the OPUS-calculus is too low level, because arguments need to be modelled in terms of private attributes, and similarly for self sends. For this reason examples become complicated very quickly.

In [Steyaert & De Meuter 95], a denotational semantics for a subset of AGORA (called MiniMix) is presented. Although this semantics was a major source of inspiration, MiniMix and AGORA only offer a limited form of encapsulated modification of objects. They only offer part of the functionality —inheritance with mixin methods— but offer no mechanism for conservative modifications. Furthermore the ability to explicitly name and manipulate the self of an object as a generator makes our inheritance mechanism more flexible than mixin methods. For example, one can access the generator of a non direct parent in our model, allowing the implementation of stub methods.

8 Conclusions and Future Work

In this paper we proposed a formal foundation for encapsulated modification of objects, taking into account a number of design guidelines we believe to be essential to OO. Two methods for incremental modification on objects without endangering encapsulation were presented. On the one hand inheritance, with late binding,

which is only possible by privileged inheriting clients “from inside” an object. On the other hand conservative modifications, without late binding, by unprivileged clients.

Inheritance tends to be implementation dependent, whereas conservative modifications are of an abstract nature since they can only access the client interface. Both strategies are needed since they serve different purposes. Omitting inheritance would be a total neglect of the power of current day OO programming techniques relying heavily on late binding of self. Prohibiting conservative modifications and only allowing inheritance is also very restrictive. Since encapsulated modification is limited to the inside of an object, any possibly needed modification would have to be implemented inside the object. This is in fact a major criticism on the AGORA language which offers only mixin methods as an incremental modification mechanism.

From a software engineering point of view the manifestation of these different kinds of modification seems very natural. Code for an implementation dependent modification of an object will have to be nested somewhere inside the object (e.g. in a nested mixin method), thus clearly identifying it as belonging to that object and depending on its implementation details. In contrast a conservative modification, which only depends on the client interface of an object can be applied to several objects sharing the same abstract functionality and can be encoded separately from the object(s).

A possible research track is the introduction of typing in the model. When trying to add typing to our model some unexpected difficulties might turn up [Abadi & Cardelli 94]. However, based on the results of [Fisher & Mitchell 95] and [Lucas et al. 95] we are confident that this will not be the case.

Also an attribute visibility mechanism should be looked at. Such a mechanism is indispensable in real programming languages. In the current paper we chose not to include a method hiding mechanism in the basic syntax to keep it as simple as possible.

9 Acknowledgements

We deeply appreciate the help of Patrick Steyaert for his numerous suggestions and remarks during the entire development process of this work. Thanks to Carine Lucas for proofreading a draft version of this paper. We are also indebted to our promotor Theo D'Hondt for supporting our research efforts.

10 References

[Abadi & Cardelli 94] Abadi, M. & Cardelli, L. - 1994. A Theory of Primitive Objects: Untyped and First-Order Systems. TACS ‘94 Proceedings, Springer-Verlag.

[Boyen et al. 94] Boyen, N.; Lucas, C. & Steyaert, P. - 1994 Generalised mixin-based inheritance to support multiple inheritance. Technical Report vub-prog-tr-94-12, presented as poster at OOPSLA ‘94.

[Bracha & Cook 90] Bracha, G. & Cook, W. - 1990. Mixin-based Inheritance. Proceedings of Joint OOPSLA/ECOOP ’90 Conference, pp. 303-311, ACM Press.

[Cardelli & Mitchell 89] Cardelli, L. & Mitchell, J. C. - 1991 Operations on Records. Mathematical Structures in Computer Science, 1(1):3-48.

[Codenie et al. 94] Codenie, W.; De Hondt, K.; D’Hondt, T. & Steyaert, P. - 1994. Agora: Message Passing as a Foundation for Exploring OO Language Concepts. SIGPLAN Notices, Volume 29, Number 12, December 1994, pp. 48-58, ACM Press.

[Cook 89] Cook W. - 1989, A Denotational Semantics of Inheritance, Ph.D.-Thesis, Brown University.

[Dony et al. 92] Dony, C.; Malenfant, J. & Cointe, P. - 1992. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. OOPSLA ‘92 Proceedings, pp. 201-217, ACM Press.

[Fisher & Mitchell 95] Fisher, K. & Mitchell J. - 1995. A Delegation-based Object Calculus with Subtyping. FCT ’95, LNCS 965, pp. 42-61, Springer-Verlag.

[Goldberg & Robson 83] Goldberg, A. & Robson, D. - 1983. Smalltalk 80: The Language and its Implementation, Addison-Wesley.

[Lamping 93] Lamping, J. - 1993. Typing the Specialization Interface. OOPSLA ‘93 Proceedings, pp. 201-214, ACM Press.

[Lucas et al. 95] Lucas, C.; Mens, K. & Steyaert, P. - 1995. Typing Dynamic Inheritance: A Trade-off between extensibility and substitutability. Submitted to TAPOS Journal.

[Mens et al. 94] Mens, T.; Mens, K. & Steyaert, P. - 1994. OPUS: a Calculus for Modelling Object-Oriented Concepts. OOIS '94 Conference, pp. 152-165, Springer-Verlag.

[Mezini 95] Mezini, M. - 1995. Supporting Evolving Objects Without Giving Up Classes. TOOLS ‘95 Conference.

[Nierstrasz 89] Nierstrasz, O. - 1989. A survey of object-oriented concepts. Object-Oriented Concepts, Databases and Applications, pp. 3-21, ACM Press and Addison Wesley.

[Pierce & Turner 93] Pierce, B. & Turner, D. - 1993. Object-Oriented Programming Without Recursive Types.

[Schmidt 86] Schmidt, D. A. - 1986. Denotational Semantics: A Methodology for Language Development; Allyn and Bacon, Inc.

[Snyder 87] Snyder, A. - 1987. Inheritance and the Development of Encapsulated Software Components. Research Directions in Object-Oriented Programming; (eds.) Shriver, B. & Wegner, P.; pp. 165-188; MIT Press.

[Steyaert & De Meuter 95] Steyaert, P. & De Meuter, W. - 1995. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. ECOOP ‘95 Proceedings, LNCS 952, pp. 127-144, Springer-Verlag.

[Steyaert 94] Steyaert, P. - 1994. Open Design of Object-Oriented Languages: A foundation for Specialisable Reflective Frameworks. Ph.D.-thesis, Vrije Universiteit Brussel.

[Steyaert et al. 93] Steyaert, P.; Codenie, W.; D’Hondt, T.; De Hondt, K.; Lucas, C. & Van Limberghen, M. - 1993. Nested Mixin-Methods in Agora. ECOOP ‘93 Proceedings, LNCS 707, pp. 197-219, Springer-Verlag.

[Stroustrup 92] Stroustrup B. - 1992. The C++ Programming Language Second Edition. Addison Wesley.

[Ungar & Smith 87] Ungar, D. & Smith, R. B. - 1987. Self: The Power of Simplicity. OOPSLA ‘87 Proceedings, pp. 227-242, ACM Sigplan Notices, Vol. 22, No. 12, ACM Press.

[Wegner & Zdonik 88] Wegner, P. & Zdonik, S. B. - 1988. Inheritance as an Incremental Modification Mechanism, or What Like is and Isn’t Like. ECOOP ‘88 Proceedings, LNCS 322, pp. 55-77, Springer-Verlag.