View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by El Servicio de Difusión de la Creación Intelectual

Formalising Sharing Mechanisms in Object-Oriented Paradigm

Verónica Argañaraz, Gabriel Baum, Claudia Pons,
María José Presso, Máximo Prieto, Natalia Romero.

LIFIA, Departamento de Informática, Facultad de Cs. Exactas,
Universidad Nacional de La Plata.

E-mail: [vero, majo, nachi] @info.unlp.edu.ar,
[gbaum, cpons, maximo] @sol.info.unlp.edu.ar

Abstract

Sharing of behavior is one of the most important features in the Object-Oriented paradigm. The two classical organisations of sharing are classes and prototypes, raising two different models and two families of object oriented languages. It has been largely discussed which of these two models is the most basic, giving the essence of the Object-Oriented paradigm.

We claim that sharing schemes can be constructed in a more basic model with just objects and messages. We analysed the features this model must express, specially the ability to share behavior. Abadi and Cardelli have defined a calculus of objects which represents the basic elements. They describe how to build the concepts of class based languages in their formalism.

In this work we show how to express delegation between concrete objects in the calculus. The key advantage of our contribution is that by providing per object delegation we can represent every sharing scheme possible in a prototype environment, thus completing the conviction that all the usual constructs found in OO can be built using only objects and messages.

We have defined sharing constructs for an object based (prototypes) high level language, and their translation into the formal calculus. This shows how constructs similar to those appearing in usual programming languages can be written in the formal calculus, and allows writing programs in the formalism without requiring understanding details about it. There are primitives to express the sharing relationship in object creation, to change that relationship and to reference the donor of an object.

Introduction

One of the most important features in the Object-Oriented paradigm is sharing. Sharing is what allows objects to have a common behavior. By sharing, an object exhibits behavior it does not define by itself, but is rather defined by some other object. There are two main ways of organising the sharing mechanism, thus raising two different models and two different families of Object-Oriented languages.

One of these models is based on the class concept. In this model, classes are objects that define behavior for a collection of other objects created by them. The objects created by a class are called instances of that class. Every object belongs to a class and borrows from it the methods needed to answer the messages it receives.

The other model is based on the prototype concept. In this model an object can be created by any other object (the prototype), being the new one a clone of the creator, and can borrow behavior from any object too.

It has been largely discussed which of these two models is better suited for development and which one is the most basic, giving the essence of the Object-Oriented paradigm. Lieberman [Lieberman86] showed how to implement sharing mechanisms between groups of objects using prototypes. Lynn Andrea Stein [Stein87] showed how to simulate prototypes with classes.

The discussion about suitability for development is quite solved since the “Treaty of Orlando” [Stein88]: each model is more appropriate for different domains and development stages. Prototypes are very flexible, and so allow rapid prototyping and exploratory programming. They are useful for domains under investigation. Classes allow good structuring of the system, and are useful when the domain is well understood, and to build reusable components.

Nevertheless, the question about which model is the most basic remains open and leads us to the question: Is there a minimal set of concepts that serves as the basis to build both, classes and prototypes? Considering that in the Object-Oriented world everything must be modelled as an object, it is reasonable to think that these concepts are objects and object communication, i.e. messages. We have selected and defined the basic concepts we consider indispensable in a model of Object-Orientation. One of these is sharing, a complex notion that must be specially analysed.

This model should fulfil some requirements as, for example:

- being formal,
- being expressive enough, so that the concepts in the Object-Oriented paradigm can be specified,
- being adequate with relation to the Object-Oriented paradigm,
- being simple enough so that the amount of mathematical or logical knowledge necessary to understand or express the basic concepts about objects is minimal,
- and being wide enough so that not only the basic concepts can be specified, but also design and programming techniques.

Over the last years there has been important effort to develop formal theories about the basic concepts in Object-Oriented programming. Most existing models of Object-Oriented languages fall into one of three categories:

Models in the first category, such as the ones defined in [Breu89, Ehrich90, Goguen94, Lu93], rest on the appealing simile between the programming concept of object and the mathematical notion of algebra.

Models of the second kind [Abadi96, America90, Bruce91, Castagna92, Cook90, Steyaert95] describe object-oriented languages using some formal notation, such as lambda calculus.

Models in the third category [Bahsoun93, Fiadeiro96, Meseguer91, Wieringa94] give formal semantics of an object as a logic theory.

Most of these models define the concept of class as a basic construction of object-oriented languages and make emphasis on the semantics of class-based inheritance. There has been less

effort to develop foundations for prototype-based languages or to define a more general model, which at the same time represents both class-based and prototype-based languages.

Martin Abadi & Luca Cardelli have defined a formal calculus of objects, the impς-calculus, which consists just of objects, object communication and object update [Abadi96]. It belongs to the second category described above. It appropriately expresses the basic notions about objects and neither classes nor prototypes are primitive in the calculus. Abadi & Cardelli describe how to build the concepts of class based languages in their formalism, but they don't represent the ability to share behavior by delegation among concrete objects present in prototype based languages.

We have built per object delegation using the calculus. The key advantage of our contribution is that by providing per object delegation we can represent every sharing scheme possible in a prototype environment, thus completing the conviction that all the usual constructs found in Object-Oriented languages can be built using only objects and messages.

We have defined sharing constructs for an object based (prototypes) high level language, and their translation into the formal calculus. This shows how constructs similar to those appearing in usual programming languages can be written in the formal calculus, and allows writing programs in the formalism without requiring understanding details about it. There are primitives to express the sharing relationship in object creation, to change that relationship and to reference the donor of an object.

The structure of the paper is as follows. Section 1 describes the Object-Oriented paradigm essentials. Section 2 discusses the notion of sharing, the different aspects it involves and the usual sharing mechanisms. Section 3 briefly describes Abadi & Cardelli imperative calculus and how it represents the essential Object-Oriented constructs. Section 4 shows the representation of per object delegation in the calculus and Section 5 gives the high level constructs for per object sharing and their translation into the calculus. Section 6 exemplifies the use of the proposed language, giving solutions to known problems. We conclude by stating some final remarks and future work.

1. Essentials

In this section we describe what we consider the essential features of the Object-Oriented paradigm.

The basic elements in the model are objects. The basic mechanisms, indispensable to work with objects are object creation, object communication and object modification. The use of the basic elements and mechanisms must exhibit the essential characteristics of polymorphism, encapsulation and ability to share. We next describe each feature.

Objects are abstractions of entities from the real world, this counts for concrete as well as for abstract entities. The fundamental capability of objects is communication. Communicating means receiving a message, and acting in response to it. Every object has a set of messages it understands, known as the object's protocol. The object knows what to do in response to each message it understands. Besides, there are messages the object does not understand. The object also knows what to do when it receives a message it does not understand.

For each message an object understands it knows another object, which gives the response to the message and is called method. The method object may describe interactions with other objects, necessary to fulfil the task commended by the message.

The set of messages an object understands and the responses to those messages are the object's behavior.

This notion of messages and associated responses make needless the usual notions of instance variables and state: an instance variable is a message whose response is the object which would be referenced by the variable. In this frame, we do not distinguish between state and behavior.

Objects have identity. The object's identity won't change, even if the object changes. Different objects are distinguishable upon their identity, no matter any other similarity between them.

Object creation is a mechanism for introducing new objects into the world. In its basic form, it is a way of generating objects from scratch: an object is defined by describing its behavior.

Object communication is achieved by messages. A message is a request to perform an action, or a

notification of an event. It produces a reaction on the object that receives it. A message may carry one or more objects with which the receiver can interact to response to the message.

We have stated that objects are entities that can receive messages and answer them. Then, modifying an object means changing the response it gives to some message. An object modification mechanism must allow to replace an object's response to a message.

The manipulation of objects with the basic mechanisms must conform certain characteristics: polymorphism, encapsulation and ability to share.

Polymorphism means that different objects may be able to answer the same message, and the reaction to that message will depend on the object that receives the message.

Encapsulation is respected if an object can be modified just by itself, in response to a message. Besides, an object can restrict the set of messages it can receive from other objects to a subset of its messages. This is, the set of messages is divided in private and public messages, and only the object itself can send it a private message.

Objects must be able to share their behavior with other objects: being created in terms of existing objects, borrowing behavior from other objects. Sharing is further analysed in the following section.

The basic elements and mechanisms just defined are the essential building blocks in the Object-Oriented paradigm. Legal constructs from this blocks must respect encapsulation and behave polimorphycally. Sharing can be accomplished by appropriate constructs from the basic elements.

2. Sharing

One of the most important features in Object-Oriented programming languages is the facilities they provide for different objects to have common behavior.

Sharing of behavior can be implicit or explicit, it can be defined per objects or per groups of objects, it can be dynamic or static and it can be implemented using delegation or embedding [Stein88, Abadi96]. These choices are orthogonal, in the sense that any combination of them is legal, and determines a different kind of sharing.

When two objects share behavior one of them must know to behave upon the reception of a message, called donor, and the other object borrows that behavior.

Sharing is explicit if an object can explicitly designate a donor for each message. In this case, an object can have many donors, one for each message for which it borrows the behavior. Sometimes this flexibility gives raise to complex and confusing sharing relationships.

Sharing is implicit if an object has a parent object from which it borrows the behavior. In this case, the parent is the default donor for each message for which the object doesn't define its own behavior. Some languages allow the definition of more than one parent object (resulting, for example, in multiple inheritance).

When sharing is dynamic an object has the ability to change its donors, to decide not to borrow behavior for a message any more and implement its own behavior, or to start borrowing it.

With static sharing the donor objects cannot be changed, and the sharing relationship is fixed from the creation.

Sharing of behavior can be accomplished with delegation or embedding.

With embedding, the borrowed behavior becomes part of the borrower object (usually the behavior is copied in the object at the time the object is created). In this case, when the borrower object receives a message for which it borrows the behavior it can give an answer by itself, and no interaction with the donor is needed.

With delegation, each time the borrower object receives a message for which it doesn't define its own behavior, it asks the donor to handle the message for it. The donor must be able to respond to the message on behalf of the receiver.

A main feature to deal with when providing implicit sharing with delegation is the routing of messages following the donor relationship. One choice is that the borrower object knows just those messages for which it implements the behavior. When an object receives a message it can not understand, the

message is forwarded to its donors. The donors would do the same if they don't understand the message. The message follows the chain until reaching an object that understands it (and responds to it), or an object that does not understand it and has no donors.

Another choice is that every object knows which are the messages for which it borrows behavior. So when it receives a message it can decide whether to forward it to a donor or just discard it. In this case the shared protocol is embedded in the borrower object.

In class based languages sharing is expressed for groups of objects (when you define behavior for a class all the objects belonging to that class borrow it) and it is implicit (each object has a parent: its own class). But some class-based Object-Oriented languages use embedding and others use delegation, in some of them sharing is dynamic, in others it is static.

Smalltalk, for example, uses a combination of embedding and delegation. Instance variables in Smalltalk are embedded (so that each object can keep its own state) but the other messages are delegated to the class [Goldberg83].

Hybrid's inheritance is dynamic [Stein88]. In C++, on the other hand, the sharing relationship is static. In Smalltalk sharing is intended to be static, even though there are messages to change an object's class, their use is discouraged.

Prototype based systems use per object sharing. But different languages choose differently whether to use embedding or delegation (or both), explicit or implicit, static or dynamic sharing. For example, some languages provide cloning, which is implicit embedding, as a mechanism for creating objects, and also provide delegation. Self implements implicit sharing through the parent link, and in Delegation language sharing can be either implicit or explicit [Stein88].

3. Presentation of impς-calculus.

In this section we describe an imperative, untyped calculus of objects, created by Abadi & Cardelli [Abadi96], and the relation to the essential notions described in Section 1. Then we explain functions and pre-methods, which are used to construct sharing of behavior in the calculus.

We choose this calculus because it provides the basis for the model we wanted. Every term in the calculus is a directly described object, the response to a message or the result of an update.

An object is a collection of messages the object understands; for each message there is a method or response to the message.

An object understanding messages $m_1, m_2, .., m_n$ with associated methods $b_1, b_2, .., b_n$ is represented by the term

$$[\ m_1 = \varsigma(x_1)\ b_1, m_2 = \varsigma(x_2)\ b_2, .., m_n = \varsigma(x_n)\ b_n\ ]$$

$b_1, b_2, .., b_n$ are themselves objects. The construct $\varsigma(x_1)\ b_1$ means that in the body b_1, the variable x_1 will represent the receiver of the message m_1 (the usual notion of self)

Sending the message m_1 to the object o, is represented by the term

$$o.m_1$$

The update of object o, setting the method for m_1 to object b_1, is represented by

$$o.m_1 \Leftarrow \varsigma(x_1)b_1$$

The creation of an object is achieved by the construction of the term representing the object.

The calculus also includes terms for a cloning operation: *clone (a)*, which creates a new object as a copy of object a (but with its own identity); and a let construction: *let x=a in b*, (here a and b are objects and x is a variable) representing the evaluation of a, followed by the evaluation of b, with x standing for the result of the evaluation of a.

Polymorphism is inherent to the calculus, because there is no restriction about two objects having the same message in their protocol, and the method for the message is always taken from the receiving object.

Sharing is achieved by the use of pre-methods. Pre-methods are objects representing functions that take an object (the receiver of the message, “self”) and return a method. Pre-methods are methods

in some other object (e.g. a class object) and they are obtained by message passing.

To achieve sharing in the calculus, the receiver must understand the message (i.e. the message must belong to its protocol), and in its associated method it must borrow the proper method sending a message to the object that has the required pre-method. We call this mechanism "protocol embedding".

Encapsulation is not directly expressed in the calculus. This characteristic is not covered in this work and will be subject of further work.

Operational Semantics

The operational semantics describes term reduction. It is based on a global store. Objects reduce to sequences of store locations, one for each of its messages. The definition uses stacks, which bind names to object results. Each store location contains a method closure, which is composed of a method and a stack used to evaluate the method.

The operational semantics relates a store σ, a stack S, a term b, a result v and another store σ'

$$\sigma . S \vdash b \;\; \sim> \;\; v . \sigma'$$

This means that with the store σ and the stack S, the term b evaluates to the result v, producing an updated store σ'.

The rules defining the operational semantics are depicted in Figure 1.

Rules (**Store ∅**), (**Store ι**), (**Stack ∅**) and (**Stack** ***x***) describe the formation of stacks and stores.

Rule (**Red x**) says that a variable reduces to the result it denotes in the current stack. Rule (**Red Object**) states that an object reduces to a result consisting of a fresh collection of locations; extending the store with the new locations. Rule (**Red Select**) says that when a message is sent to an object, the object is reduced to a result, and then the corresponding method is evaluated. An object update, (**Red Update**), reduces its object to a result and updates the appropriate store location with a new method closure. A cloning operation, (**Red Clone**), reduces its object to a result and then allocates a fresh collection of locations that are associated to the existing method closures from the object. A let construct is evaluated by first reducing the term associated to the bound variable to a result, and then evaluating the body in a stack extended with the variable denoting this result (**Red Let**).

Functions

Functions are represented in the calculus by objects. They are important because they are is used to represent messages that carry arguments, and to represent pre-methods. Pre-methods are used for sharing of behavior.

The representation of a function is an object with two messages: one for the argument and another for the body. Within the body, the argument is referenced by sending to self the message that returns the parameter.

The evaluation of the function involves modifying the object by setting the argument's message to return the real parameter, and then sending the message associated with the body.

For example, let's represent a function with parameter x, which calculates $x+2$. Assume that the parameter object understands the message succ, which returns the object $x+1$. This function is represented by the object

$$[\ arg = \varsigma(x_1)\ [\ ],\ val = \varsigma(x_2)\ ((x_2.\ arg).\ succ).\ succ\ ]$$

Here, the value of the parameter is initially set to the empty object. The evaluation of the above function on an object o, is represented by the term

$$(\ [arg = \varsigma(x_1)\ [\ ],\ val = \varsigma(x_2)\ ((x_2\ .\ arg).\ succ).\ succ\ ].\ arg \Leftarrow \varsigma(x_1){:}o\ ).\ Val$$

ι	Store location(e.g. an integer value)
$v ::= [l_i = \iota_i{}^{i\in 1..n}]$	result
$\sigma ::= \iota_i \rightarrow <\varsigma(x_i)b_i\ ,\ S_i>^{\ i\ \in 1..n}$	store
$S :: x_i \rightarrow v_i{}^{\ i\ \in 1..n}$	stack
$\sigma \vdash <>$	well-formed store judgment
$\sigma\ .\ S \vdash <>$	well-formed stack judgment
$\sigma\ .\ S \vdash a \sim> v\ .\ \sigma'$	term reduction judgment

(**Store** $\varnothing$)

$$\frac{}{\varnothing \vdash <>}$$

(**Store** ι)

$$\frac{\sigma\ .\ S \vdash <> \quad \iota \notin dom(\sigma)}{\sigma,\ \iota \rightarrow <\varsigma(x)b\ ,\ S> \vdash <>}$$

(**Stack** $\varnothing$)

$$\frac{\sigma \vdash <>}{\sigma\ .\ \varnothing \vdash <>}$$

(**Stack** x) (l_i, ι_i distinct)

$$\frac{\sigma\ .\ S \vdash <> \quad \iota_i \in dom(\sigma) \quad x \notin dom(S) \quad \forall i \in 1..n}{\sigma\ .\ (S,\ x \rightarrow [l_i\ = \iota_i{}^{\ i\in 1..n}]) \vdash <>}$$

(**Red** x)

$$\frac{\sigma\ .\ (S',\ x \rightarrow v,\ S'') \vdash <>}{\sigma\ .\ (S',\ x \rightarrow v,\ S'') \vdash x \sim> v\ .\ \sigma}$$

(**Red Object**) (l_i , ι_i distinct)

$$\frac{\sigma\ .\ S \vdash <> \quad \iota_i \notin dom(\sigma)\ \ \forall i \in 1..n}{\sigma\ .\ S \vdash [l_i = \varsigma(x_i\ )b_i\ ] \sim> [l_i\ = \iota_i{}^{\ i\ \in 1..n}\ ]\ .\ (\sigma, \iota_i\ \rightarrow\ <\varsigma(x_i\ )b_i\ ,\ S>^{\ i\in 1..n}\ )}$$

(**Red Select**)

$$\frac{\sigma\ .\ S \vdash a \sim> [l_i\ = \iota_i{}^{\ i\ \in 1..n}\ ]\ .\ \sigma' \quad \sigma'(\iota_j) = <\varsigma(x_j\ )\ b_j\ ,\ S'> \quad x_j\ \notin dom(S')\ \ j \in 1..n \qquad \sigma\ .\ (S',\ x_j \rightarrow [\ l_i = \iota_i{}^{\ i\ \in 1..n}\ ]\ ) \vdash b_j\ \sim> v\ .\ \sigma''}{\sigma\ .\ S \vdash a.l_j\ \sim> v\ .\ \sigma''}$$

(**Red Update**)

$$\frac{\sigma\ .\ S \vdash a \sim> [l_i = \iota_i{}^{\ i\ \in 1..n}\ ]\ .\ \sigma' \quad j \in 1..n \quad \iota_j \in dom(\sigma')}{\sigma\ .\ S \vdash a.l_j\ \Leftarrow \varsigma(x)b \sim>\ [l_i\ = \iota_i{}^{\ i\ \in 1..n}\ ]\ .\ (\sigma'.\ \iota_j \leftarrow <\varsigma(x)b\ ,\ S>}$$

(**Red Clone**) ι_i' distinct

$$\frac{\sigma\ .\ S \vdash a \sim> [l_i\ = \iota_i{}^{\ i\ \in 1..n}\ ]\ .\ \sigma' \quad \iota_i\ \in dom(\sigma') \quad \iota_i' \notin dom(\sigma')\ \ \forall\ i \in 1..n}{\sigma\ .\ S \vdash clone(a) \sim>\ [l_i\ = \iota_i'{}^{\ i\ \in 1..n}\ ]\ .\ (\sigma',\ \iota_i' \rightarrow \sigma'(\iota_i\ )^{\ i\ \in 1..n}\ )}$$

(**Red Let**)

$$\frac{\sigma\ .\ S \vdash a \sim> v'\ .\ \sigma' \quad \sigma'.\ (S,\ x \rightarrow v') \vdash\ \ b \sim> v''\ .\ \sigma''}{\sigma\ .\ S \vdash let\ x{=}a\ in\ b \sim> v''\ .\ \sigma''}$$

Figure 1

For writing convenience, functions are included in the syntax of the calculus, and translated as described above. A function with argument x and body b is written $\lambda(x)\ b$. The application of a function f to an argument a is written $f(a)$.

Pre-Methods

Pre-methods are objects used to share behavior. Pre-methods are the basis for constructing classes [Abadi96] and delegation, as we will describe in the following section. A pre-method is an object representing a function where the intended meaning of the parameter is the receiver of the message. The body refers to the parameter everywhere it would refer to self. The pre-method is a method in some object, which will be the donor in the sharing relationship. The object that wants to inherit/reuse the behavior defined in the donor, sends the donor the message for the pre-method with itself as parameter.

Suppose we have an object representing a bank account:

$$[balance = \varsigma(self)\ b\ ,$$
$$deposit = \varsigma(self)\ \lambda(amount)\ let\ x = self.\ balance.\ add\ (amount)\ in\ (\ self.\ balance \Leftarrow \varsigma(self)\ x\ ),$$
$$withdraw = \varsigma(self)\ \lambda(amount)\ let\ x = self.\ balance.\ sub\ (amount)\ in\ (\ self.\ balance \Leftarrow \varsigma(self)\ x\ )\ ]$$

Now we want to share the behavior for depositing and withdrawing between several bank account objects. We will create a new object that abstracts this behavior in a way it can be borrowed by the particular accounts. Let's call *abs-account* this new object.

$$abs\text{-}account \equiv$$
$$[deposit = \varsigma(self)\ \lambda(receiver)\ \lambda(amount)$$
$$let\ x = recevier.balance.add\ (amount)\ in\ (recevier.balance \Leftarrow \varsigma(self)\ x\ ),$$
$$withdraw{:}\varsigma(self)\ \lambda(receiver)\ \lambda(amount)$$
$$let\ x = recevier.balance.sub\ (amount)\ in\ (recevier.balance \Leftarrow \varsigma(self)\ x\ )]$$

Now a particular account will be represented by

$$[balance = \varsigma(self)\ b,$$
$$deposit = \varsigma(self)\ abs\text{-}account.deposit(self),$$
$$withdraw = \varsigma(self)\ abs\text{-}account.withdraw(self)]$$

4. Implementing implicit delegation in impς-calculus

In an Object-Oriented environment with implicit delegation every object has a donor.

The imperative impς-calculus of Abadi & Cardelli does not provide implicit delegation as a primitive mechanism. The main problem to simulate implicit delegation arises because objects only perform actions when they receive **known** messages; the reception of unknown messages produces a stuck state in the reduction. In this representation objects can only receive messages for which they have a response.

Although implicit delegation is not the implementation of sharing chosen by Abadi & Cardelli, it can be simulated at a user level language with impς-calculus as its kernel.

Implicit donor is easily implemented with the more general concept of explicitness provided by the calculus. If we have a way to specify the donor for a whole object, we can make explicit references to the same donor in the object's methods, in order to get the effect of implicit donor.

To give delegation semantics we have to analyse and simulate the two defining characteristics, mentioned in Section 2.

The first of them is the capability of a donor to respond to messages as if it were the receiver.

We have to simulate the handling of self in shared methods. The idea of pre-methods is the key.

Recall that pre-methods are methods whose self variable is abstracted. If we want to implement an environment where every object can be a donor for others (any object can be taken as a prototype, and prototypes are ordinary objects) we have to implement methods in objects with a pre-method style.

The other defining characteristic of implicit delegation is the routing of messages following the donor relation. Due to the restricted capability of objects in the impς-calculus to receive only known messages, this routing cannot be emulated. Instead, its effect can be achieved through embedding of protocol. When an object is created it is prepared to delegate in a specified parent object: its protocol is extended with a copy of the donor's protocol. In this way it is capable of responding to any of the shared methods. Messages are represented with a specification part and an implementation part. The former is a message whose method is an invocation to the implementation part. The latter is another message whose method is the pre-method. The specification part invokes the corresponding pre-method with the receiver object as argument. The implementation part contains the body of the method for the messages defined by the object, or an invocation to the parent's corresponding pre-method for borrowed messages.

So, we could have the following construct in a high level language:

object *obj* **is**
  parent *donor*
  $message_i(arg_{i,1},..,arg_{i,pi}) : method_i\ ^{i\ \varepsilon\ 1..n}$
end

to specify that *obj* is a new objet being created. Object *donor* is set as *obj*'s parent. Messages $message_1$ to $message_n$ are the new original messages of *obj,* with $arg_{i,j}$ their arguments.

If we suppose that *donor* is capable of receiving messages $message_{n+1}$ to $message_{n+m}$ disjoint with $message_1,..,message_n$, (note that the variation to allow overriding of messages is straightforward), the translation of the previous constructs into impς-calculus could be:

$$\begin{aligned}
\text{let } obj = [\ & \text{parent} : \varsigma(\text{self})\ donor, \\
& message_i : \varsigma(\text{self})\ \text{self}.{*message_i}(\text{self})\ ^{i\varepsilon 1..n+m}, \\
& {*message_i} : \varsigma(\text{self})\ \lambda(\text{rec})\ \lambda(arg_{i,1}). . .\lambda(arg_{\ i,pi})\ method_{\ i}\{arg_{\ i,j}\ ^{j\ \varepsilon\ 1..pi}\}\ ^{i\varepsilon 1..n}, \\
& {*message_i} : \varsigma(\text{self})\ \lambda(\text{rec})\ \text{self.parent}.{*message_i}(\text{rec})\ ^{i\ \varepsilon\ n+1..n+m}\ ]
\end{aligned}$$

The special symbol '*' is used to denote messages containing pre-methods. Messages without '*' correspond to the specification parts, observe that *obj* contains all the messages borrowed from *donor* among them. These messages without '*' invoke the corresponding pre-method. The responses for *obj*'s proper messages ($message_1$ to $message_n$), $method_1$ to $method_n$, are contained in the corresponding *obj*'s pre-methods ($*message_1$ to $*message_n$). The implementation for borrowed messages ($message_{n+1}$ to $message_{n+m}$), is an invocation to pre-methods $*message_{n+1}$ to $*message_{n+m}$ in *donor.*

Any object responding to the same protocol as *donor* could be set as *obj*'s parent at run-time, replacing *donor* through a high level sentence such as:

obj **setParent:** *donor2*

which could be translated into impς-calculus as

obj.parent $\Leftarrow \varsigma$(self) *donor2*

In this way we get a clean implementation of implicit delegation constructed over a calculus that does not provide it as primitive. The user level language could be extended in order to provide multiple

parent and some form of explicit delegation (delegating particular messages to the donor within an original or overridden method, in a taste to *super*).

5. An object based language providing sharing

In this section we present Asterix, a simple object based user level language that provides several ways of expressing sharing. Fixing a per object model, it has constructs for both delegation and embedding, and these can be used in an implicit or explicit form. The translation of Asterix objects to impς-calculus terms shows how the different sharing schemes can be built upon the calculus. As we use a formalism -the impς-calculus- to give the semantics of our high level language, we obtain a rigorously defined user language.

Objects in Asterix are prototypes: they can be cloned to create new objects, and other objects can delegate behavior in them.

The attributes of objects are treated homogeneously. There is no distinction between fields and methods.

There are two main ways of creating objects in the language: defining and cloning. Object definition generates a new object specifying the messages it receives and optionally a parent object. The sharing schemes for an object's messages are determined at creation time, and can be later modified. The new object delegates in its parent all the messages it does not define. Messages defined for the new object may override parent's or may be new proper messages. The method for a message may be defined from scratch (optionally with arguments), or may be borrowed in three different ways: delegating in an existing object, embedding it from its parent, or from any other object.

object
  [**parent:** *o1*]
  message m1 ((arg_1**,..,** arg_n) x_1,..,x_m *b* **end** |
    delegate in *o2* /
    embed from *o3* /
    embed from parent)
    message m2 . . .
  :
end

When defining a method, variable identifiers (x_1 to x_m, in the example above) may be specified to be used in the body of the method.

The cloning construct creates a new object embedding the whole behavior of an existing one.

clone(*obj*)

When an object is cloned, all the responses to its messages are copied to generate a new object. Embedding is thus done implicitly, because there is no specification of particular methods to embed, the whole object is copied.

At object creation time we have the possibility to use four different patterns of sharing. If the new object specifies a donor object with **parent:**, then every message it does not define (when received) will be forwarded to that object; in this way we have implicit delegation.

For a method defined with **delegate in** *o1*, when the corresponding message is received it is forwarded to *o1*, which acts as the donor for that specific method. So it's an explicit delegation kind of sharing. When the response to a message is defined with **embed from** (**parent**|*o1*), a copy of the corresponding method in the parent or *o1* is obtained and set as the new method. As a specific method is embedded, we classify it as explicit embedding.

To provide object modification Asterix has an update construct to change the method for a message, and a special construct to change the donor of an object. Modification of an object can only be done by the same object in the body of its methods; this way encapsulation is not violated. Method update allows modifying the sharing relationship between objects. This can be done either defining a new method, or delegating the method in the parent or in any other object, or obtaining a copy of another

object's method through embedding.

```
self m1 := ( (arg1,.., argn) x1,..,xm b end |
             delegate in o2               /
             delegate in parent           /
             embed from o3                /
             embed from parent            )
```

The parent of an object can be replaced by another object, and all the messages not defined for the receiver object will now be delegated in the new parent. That new donor should have the same protocol as the replaced one.

self changeParent: *o2*

There is also a form of delegating messages in the parent object inside the body of a proper method.

super m1[(*o2*,..,*on*)]

Besides the constructs for sharing there are other constructs that complete the language. These are: message sending, which can specify argument objects; sequence, which gives a way of structuring bodies of methods; and assignment, to bind variables to objects.

The complete Asterix syntax is:

```
o1, o2,.., on ::=
   clone(o1)                                          |  cloning
   object                                                direct object construction
       [parent: o1]                                      parent object definition
       message m1 ( (arg1,.., argn) x1,..,xm b end |     method definition
                    delegate in o1               /
                    embed from o1                /
                    embed from parent            )
       message m2 . . .
       :
     end                                              |
   o1 m1[(o2,..,on)]                                     message sending

b ::=                                                    method body object
   o1                                                 /  object
   xi, argi                                           |  variable, argument identifier
   self m1 := (arg1,.., arg n) x1,..,xm b end         |  method update
   self m1 := delegate in o1                          |
   self m1 := delegate in parent                      |
   self m1 := embed from o1                           /
   self m1 := embed from parent                       |
   self changeParent: o2                              |  parent object update
   super m1[(o2,..,on)]                               |
   b m1[(o2,..,on)]                                   |  message sending
   x := o1                                            |  variable assignment
   o1. o2                                                sequence
```

Translation of Asterix constructs into impς-calculus :

To explain the representation of Asterix constructs in impς-calculus we use a relation trans, which maps Asterix constructs to impς-calculus terms.

The translation uses the technique of pre-methods and implicit delegation introduced in Section 4. Every object has its methods implemented as pre-methods, so that any object can be used as a prototype. The only exception is the parent attribute, which keeps a reference to the parent object. The specification part of a message has the name of the message, and the implementation part is a pre-method whose name is the name of the corresponding message preceded by the special character '*' (which is not allowed in message names). The pre-method contain the body of the

method if it is proper, or specify how it is borrowed otherwise.

The impς-calculus translation of an object directly created with a proper message will have two messages, one representing the specification part of the message and the other representing the implementation. The latter contains the pre-method, which abstracts the receiver object and defines the body of the method. The body is represented as a λ-function that abstracts the arguments of the message. So the construct

object
  message m (arg_1,..,arg_n) x_1,..,x_m *b* **end**
end

is translated as

[m= ς(self) self.*m(self)
  *m= ς(self) λ(rec) λ(arg_1)...λ(arg_n) [x_1 : ς(bself) [ ]
  :
  x_m : ς(bself) [ ]
  val : ς(bself) trans(*b*){x_i← bself.x_i $^{i\varepsilon 1..m}$}]. val

Where the variable identifiers (x_1,..,x_m) defined for the method m are implemented as messages, and used within the object representing the body of the method.

The impς-calculus representation for an object with a parent and without definition of new messages, will have a message for the parent reference and will embed the parent protocol. That is, it will have a specification message for every message in its parent protocol, and a corresponding implementation message containing a pre-method which will invoke the matching pre-method in the parent, sending the receiver object (referenced by self) as the receiver argument. The translation for

object
  parent: *o1*
end

is thus

let par= trans(*o1*) in
  [parent= ς(self)par,
  mi= ς(self) self.*mi(self) $^{i\varepsilon Par}$
  *mi= ς(self) self.parent.*mi $^{i\varepsilon Par}$]

If the creation of an object defines a parent object and a new message its translation into impς-calculus will include, besides the embedding of the parent's protocol, two more messages representing the specification and implementation parts of the defined message. Thus, the object creation

object
  parent: *o1*
  message mj (arg_1,..,arg_n) x_1,..,x_m *b* **end**
end

is expressed in the calculus as

let par= trans(*o1*) in
  [parent= ς(self)par,
  mj= ς(self) self.*mj(self)
  mi= ς(self) self.*mi(self) $^{i\varepsilon Par}$
  *mj= ς(self) λ(rec) λ(arg_1)...λ(arg_n) [x_i : ς(bself) [ ] $^{i\ \varepsilon\ 1..m}$,
  val : ς(bself) trans(*b*){x_i← bself.x_i $^{i\varepsilon 1..m}$}]. val
  *mi= ς(self) self.parent.*mi $^{i\varepsilon Par}$]

When an object is created specifying a parent and overriding one of its messages, the translation will

embed all of the parent's protocol except the redefined message. It will have the corresponding two messages containing the method and pre-method for the overridden message. An object creation

object
 parent: *o1*
 message m1 (arg_1,.., arg_n) x_1,., x_m *b* **end**
end

is thus translated into impς-calculus as

let par= trans(*o1*) in
 [parent= ς(self)par,
 m1= ς(self) self.*m1(self)
 mi= ς(self) self.*mi(self) $^{i\varepsilon Par-\{1\}}$
 *m1= ς(self) λ(rec) λ(arg_1)...λ(arg_n) [x_i : ς(bself) [ ] $^{i \varepsilon 1..m}$,
 val : ς(bself) trans(*b*){x_i← bself.x_i $^{i\varepsilon 1..m}$}]. val
 *mi= ς(self) self.parent.*mi $^{i\varepsilon Par-\{1\}}$]

The representation for the direct creation of an object that explicitly delegates a message in another object, will invoke the corresponding pre-method of that object. This invocation is made in the corresponding pre-method of the message. If the object also defines a parent, the embedding of its protocol remains as before. The translation for

object
 parent: *o1*
 message mj **delegate in** *o2*
end

in the calculus is

let par= trans(*o1*) in
 let aDonor= trans(*o2*) in
 [parent= ς(self)par,
 mj= ς(self) self.*mj(self)
 mi= ς(self) self.*mi(self) $^{i\varepsilon Par}$
 *mj= ς(self) aDonor.*mj
 *mi= ς(self) self.parent.*mi $^{i\varepsilon Par}$]

If the new object defines a message embedding the response from another object, the translation will first evaluate the borrowed method in a let structure to get a new reference to it. The pre-method for the message will contain the method by using the reference obtained. So, the construct

object
 parent: *o1*
 message mj **embed from** *o2*
end

is translated as

let par= trans(*o1*) in
 let o2mj=trans(*o2*).*mj in
 [parent= ς(self)par,
 mj= ς(self) self.*mj(self)
 mi= ς(self) self.*mi(self) $^{i\varepsilon Par}$
 *mj= ς(self) o2mj
 *mi= ς(self) self.parent.*mi $^{i\varepsilon Par}$]

If the object whose method is being embedded is the parent, the translation is similar but the let structure evaluates the pre-method of the object being set as the parent

Asterix cloning construct is directly translated as a clone in the calculus. For example, the translation for

clone(*o1***)**

is

clone(trans(*o1*))

The translation of the modification of an object's response to a message modifies the implementation part of the message that is, the corresponding pre-method. In Asterix an update is allowed only inside the body of a method, so its translation will appear inside the pre-method corresponding to the method containing the update. Also, as an update in the language can only modify the self object, its representation in the calculus will always modify the receiver object in the pre-method.

If inside the method for a message m1 of an object there is a modification that replaces the response of another message m2 of the same object for a new defined method, such as

```
object
  :
  message m1 . . .
      :
      self m2 := (arg1,.., argn ) x1,., xm b end
      :
  end
  :
end
```

the translation of the update will appear inside the pre-method corresponding to m1, which will have its receiver argument modified by the update on the pre-method of m2 :

```
[ . .
  :
  *m1 : ς(self) λ(rec). .    . . rec.*m2 ⇐ ς(self) λ(rec) λ(arg1)...λ(argn)
                                              [ xi : ς(bself) [ ]  i ε 1..m ,
                                                val : ς(bself) trans(b){xi← bself.xi iε1..m}].val  . .
  :
]
```

The new pre-method has the ς-binder and the receiver argument, as usual in pre-methods, and the translation of the new defined method, with its arguments and local variables.

If the sharing scheme for a message is changed, delegating the new response in another object, such as

```
message m1 . . .
  :
  self m2 := delegate in o1
  :
end
```

the translation into the calculus is similar to the previously explained, but the replacing pre-method invokes the matching pre-method in the specified donor for the message:

```
*m1 : ς(self) λ(rec). .     . . let aDonor=trans(o1) in (rec.*m2 ⇐ ς(self) aDonor.*m2) . . .
```

If the new method delegates in the parent of the object

```
message m1 . . .
  :
  self m2 := delegate in parent
  :
end
```

a similar impς-calculus representation results:

*m1 : ς(self) λ(rec). .   . . rec.*m2 ⇐ ς(self) rec.parent.*m2 . . .

When the sharing scheme for a message is set to embedding of another object's method, such as

```
message m1 . . .
  :
  self m := embed from o1
  :
end
```

the translation into the calculus needs to first evaluate the embedded method in a let structure, and then put the new reference to it as the body of the pre-method corresponding to the updated message:

*m1 : ς(self) λ(rec). .   . . let o1m2=trans(o1).*m2 in (rec.*m2 ⇐ ς(self) o1m2) . . .

If the method is embedded from the parent of the modified object

```
message m1 . . .
  :
  self m := embed from parent
  :
end
```

a similar representation results:

*m1 : ς(self) λ(rec). .   . . let pm2=rec.parent.*m2 in (rec.*m2 ⇐ ς(self) pm2) . . .

The translation of parent modification is made in a similar way to the above modification translations. The parent object needs to be evaluated first, so as to have a proper reference to it in the parent attribute of the object.
If a method replaces the parent

```
message m1 . . .
  :
  self changeParent: o1
  :
end
```

the corresponding representation in the calculus is

*m1 : ς(self) λ(rec). .   . . let par=trans(*o1*) in (rec.parent ⇐ ς(self) par). . .

A super construct is also allowed only in the body of methods.

```
message m1 . . .
  :
  super m2
  :
end
```

Thus, its translation will also appear in the pre-method of the method containing it. To give the semantics of super, the translation uses an invocation to the corresponding pre-method of the parent of the receiver, passing the same receiver object as the receiver argument to the pre-method.

*m1 : ς(self) λ(rec). .   . . rec.parent.*m2(rec). . .

6. Examples

We will now show examples of uses of Asterix and its delegation mechanisms. We consider two known modelization problems. The problems and their solutions in class based environments are described in [Gamma95] by Decorator and State patterns. For each of them we summarise the problem and propose an alternative solution using our object based language to the example

situation modelled in [Gamma95].

The problem addressed by Decorator is adding responsibilities dynamically and transparently to individual objects, and being able to eliminate this properties.

For example, in building interfaces, we may want to add widgets to a visual component. Suppose we have a TextView object that displays text in a window. The TextView doesn't provide scroll bars by default. But we could wish to add scroll bars to the TextView, as well as some other properties like a border or a rule.

In Asterix, we solve this using delegation. We create an object whose parent is the object to be enhanced. It defines its own messages for the added responsibilities, it implicitly delegates in its parent the messages to be managed by the original object, and it redefines the messages whose behavior is changed or expanded, using super to reference its parents behavior. Multiple properties can be added repeating the procedure described. Figure 2 depicts the solution

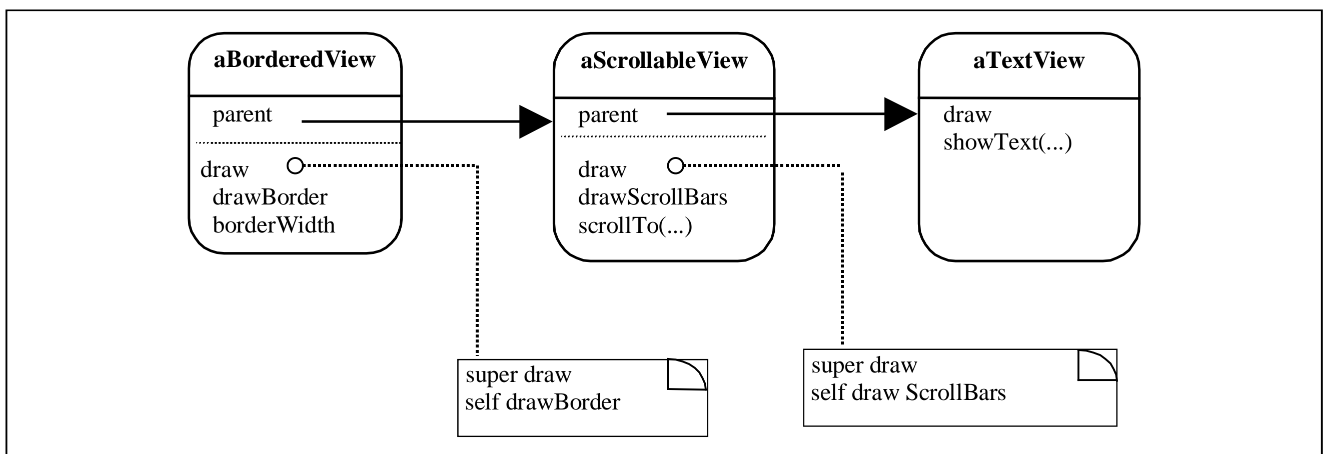

Figure 2

The Asterix code for this example is

```
object
    message body
        aTextView, aScrollableView, aBorderedView
        aTextView:= object
                message draw
                    |code for drawing|
                end
                message showText
                    |code for writing/
                end
            end .
        aScrollableView:= object
                parent: aTextView
                message draw
                    super draw
                    self drawScrollBars
                end
                message drawScrollBars
                    | code for drawing scrollBars/
                end
                message scrollTo
                    | code for scrolling |
```

```
                    end
                 end .
   aBorderedView:=object
                  parent: aScrollableView
                  message draw
                       super draw
                       self drawBorder
                  end
                  message drawBorder
                    | code for drawing border |
                  end
                  message borderWidth
                    defaultBorderValue
                  end
                  message borderWidth (aWidth)
                    self borderWidth := aWidth
                  end
               end .
  . . .
     end           | of message body |
end  |of program object|
```

The problem concerning State is the modelling of an object whose behavior depends on its state, and it must dynamically modify the way it responds to messages if that state changes.

Following [Gamma95] we will exemplify the problem with a TCPConnection that represents a network connection, which can be in one of several states: Established, Listening, Close. The TCPConnection is capable of receiving requests ActiveOpen, PassiveOpen, Close, Transmit, Send, and ProcessOctet, which open the connection, close the connection, start a transmission and transmit data. The response to each of the requests depends on the current state of the connection. Some of the responses change the current state of the connection.

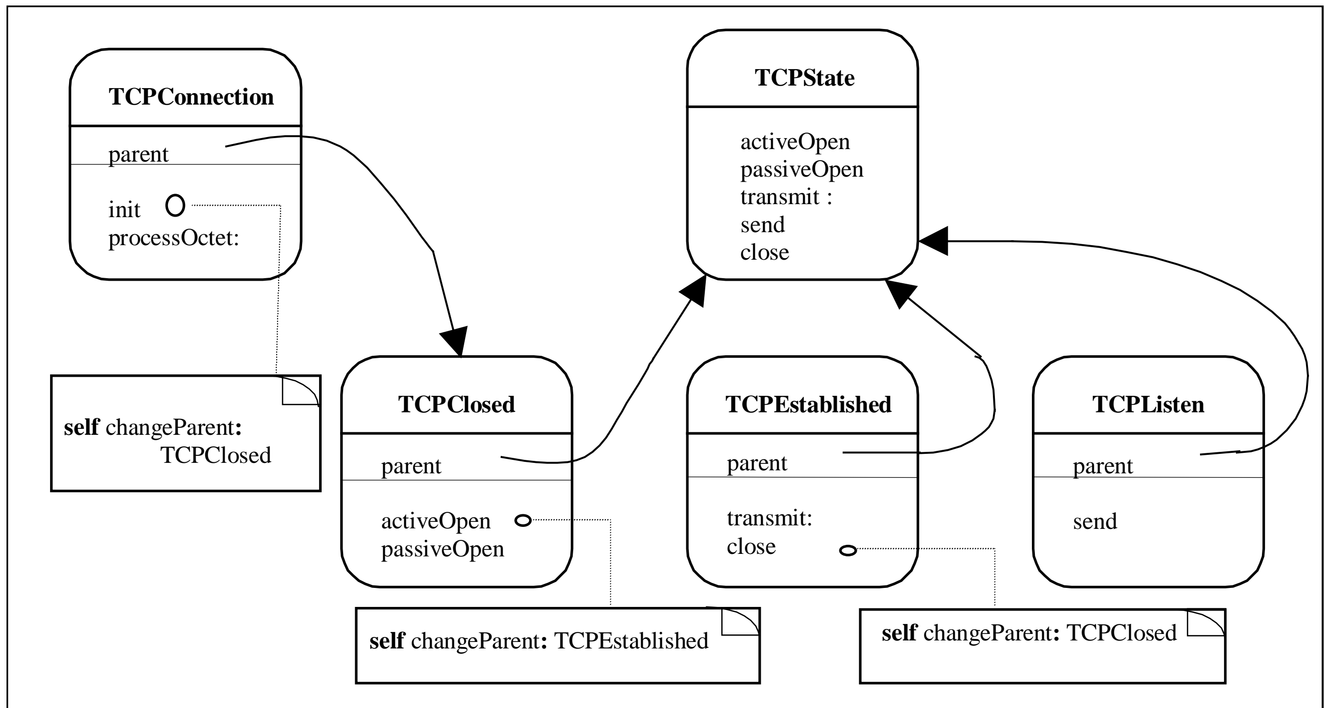

Figure 3

The design solution proposed in [Gamma95] for a class based scheme is the State pattern. We will model an alternative solution with our object based language using the delegation mechanisms it provides. TCPConnection's delegation of requests will be implemented in Asterix with the parent relation: the connection will have a state object as its parent and thus will implicitly forward all the received messages to it.

We define a general connection state object, whose protocol contains the above connection messages, to represent the default behavior of states. We also define three state objects corresponding to Established, Listening and Close states. Each of them has the general connection state as parent and redefines messages corresponding to valid requests in the state, to give the state-specific behavior. We define an additional Init message for the connection, to start working with it. Figure 3 describes the solution.
The code in Asterix is:

```
object
   message body
         TCPState, TCPEstablished, TCPListen, TCPClosed, TCPConnection, . . . |variable declarations|
         TCPState := object
                  message activeOpen
                     object end
                  end
                  message passiveOpen
                     object end
                  end
                  message transmit (octetStream)
                     object end
                  end
                  message send
                     object end
                  end
                  message close
                     object end
```

```
                end
            end .
TCPEstablished :=object
                parent:  TCPState
                message transmit (octetStream)
                    self processOctet (octetStream)
                end
                message close
                    self changeParent: TCPClosed
                end
            end .
TCPListen :=object
            parent:  TCPState
            message send
                self changeParent: TCPEstablished
            end
        end .
TCPClose :=object
            parent:  TCPState
            message activeOpen
                self changeParent: TCPEstablished
            end
            message passiveOpen
                self changeParent: TCPListen
            end
        end .
TCPConnection :=object
                parent: TCPState
                message init
                    self changeParent: TCPClosed
                end
                message processOctet (octetStream)  | implementation |  end
            end .
. . .
    end            | of message body |
end            | of program object |
```

Observe that changes of state are performed by state objects on behalf of the connection in delegated messages (with changeParent:), there is no need for message exchange as there is in the class-based approach presented in [Gamma95], where the connection class must define a 'protected' operation and the state class must be defined as 'friend' to allow it access to the operation.

7. Conclusions and future work

Over the last years there has been important effort to develop formal theories about the basic concepts in object-oriented programming. However a more solid foundation is necessary, because there is no widespread agreement on a set of basic features that characterises Object–Oriented languages. Most of the existing models define the concept of class as a basic construction of Object-Oriented languages and make strong emphasis on semantics of class-based inheritance. There has been less effort to develop foundations for prototype-based languages or to define a more general model, which at the same time represents both class-based and prototype-based languages.

We have described the basic features a model for the Object-Oriented paradigm must have, and the essential characteristics to be represented in it. We specially analysed sharing, distinguishing the different aspects that are combined to give raise to several sharing schemes.

We presented the Abadi & Cardelli impς-calculus which formally represents the above mentioned basic features and showed how to construct implicit delegation upon the calculus. Then we described a high level object based language with constructs to express every per object sharing scheme, and its translation into the calculus.

Our proposal helps to clarify the seminal concepts of Object-Oriented languages (i.e. objects, messages), shows how to interpret existing mechanisms (i.e. implicit delegation) and how to create and understand new ones (i.e. mixing explicit and implicit delegation) in a single framework.The formal translation we define provides a way for reasoning about the language (i.e. soundness proofs).

We conclude that the selected basic features are powerful enough to express the analysed sharing schemes.

In this version the model does not allow adding or removing messages from objects. The restriction is inherited from the calculus. We will work on providing this facility in our model

An essential characteristic we still don't cover completely is encapsulation. The impς-calculus is not encapsulated and encapsulation must be described in a higher level. Asterix provides encapsulation with respect to object modification, allowing it just inside the modified object. The other topic concerning encapsulation is the ability to hide certain messages. We shall work on this latter topic.

An emerging concept in the area is the notion of subjectivity, which is the ability of an object to exhibit different behavior depending on forces such as internal state, sender of a message and context. Subjectivity is now solved by modelling and programming techniques. We intend to define it in our model as a characteristic of objects, defining it rigorously in terms of the calculus.

We are currently working on tools for supporting the model, allowing working with it in a friendly way. We will construct an interpreter for the calculus and a compiler of Asterix to the calculus. Our intention is to manipulate the formal definitions as easily as we manipulate objects in usual Object-Oriented environments.

We are specially interested in the properties useful to express design quality. One of them is the notion of object types, understood as object protocols. Another is the characterization of standard collaboration relationships. The final goal is to obtain a comprehensive model and a tool to be integrated in a development environment. The tool should allow to formalise chosen parts of a design in order to reason rigorously about them. It should also allow to define and analyse typing characteristics. The tool is aimed at helping in the analysis and improvement of software design.

8. References

[Abadi96] M. Abadi and L. Cardelli, A Theory of Objects, Monographs in Computer Science, Springer, 1996.

[America90] P. América. A Layered Semantics for a Parallel Object-oriented Language. In Proceedings of Foundations of Objects Oriented Languages, LNCS 489, June 1990.

[Bahsoun93] Jean Bahsoun, S. Merz, A framework for programming and formalising concurrent objects, Software Engineering Notes, V.18, N.5, December 1993.

[Breu89] R. Breu and E. Zucca. An algebraic compositional semantics of an object-oriented notation with concurrency, LNCS 405, 1989.

[Bruce91] Kim Bruce. The Equivalence of Two Semantic Definitions for Inheritance in Object-Oriented Languages. In 8th International Conference Mathematical Foundations of Programming Semantics, USA, Proceedings LNCS 598, March 1991.

[Castagna92] G. Castagna, G. Ghelli, G. Longo, A Calculus for Overloaded functions with subtyping. In ACM Conference on LISP and Functional Programming, 1992.

[Cook90] William Cook, Walter Hill and Peter Canning. Inheritance is not subtyping. In Proc 17th ACM Symposium on Principles of Programming Languages, January 1990.

[Ehrich90] H. Ehrich, J. Goguen, A. Sernadas. A categorial theory of Objects as observed processes. In proceedings of Foundations of o-o programming, LNCS 489, 1990.

[Fiadeiro96] J. Fiadeiro, Temporal Specification of Objects, In S.Goldsack and S.Kent editors, Formal methods and object technology, chapter 9, Springer, 1996.

[Gamma95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns Elements of Reusable Object-Oriented Software, Professional Computing Series, Addison Wesley 1995.

[Goguen94] J. Goguen and R. Diaconescu, Towards an algebraic semantics for the object-oriented paradigm. In Recent Trends in Data Type Specifications, Springer-Verlag, LNCS 785, 1994.

[Goldberg83] A. Goldberg and D. Robson. Smalltalk 80: The language and its implementation, Addison-Wesley, 1983.

[Lieberman86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In Proceedings OOPSLA, 1986.

[Lu93] Xue-Miao Lu and Tharam Dillon. Towards an Algebraic Theory of Inheritance in Object Oriented Programming, AMAST'93, Proceedings. Workshops in Computing. Springer-Verlag, June 1993.

[Meseguer91] J. Meseguer, T. Winkler. Parallel Programming in Maude. Proceedings of Research Directions in High Level Parallel Programming Languages, June 1991. Springer-Verlag.

[Stein88] Lynn Andrea Stein, Henry Lieberman, David Ungar. A Shared View of Sharing: The Treaty of Orlando. In Object-0riented concepts, applications and databases, W. Kim and F. Lochowsky, eds., 31-48. Addison-Wesley, 1988.

[Stein87] Lynn Andrea Stein. Delegation is Inheritance. In OOPSLA ’87

[Steyaert95] P. Steyaert and W. De Meuter, A marriage of class- and object-based inheritance without unwanted children. In proceedings of ECOOP’95, LNCS 952, 1995.

[Ungar91] David Ungar, Randall Smith, Self: The power of simplicity. In Lisp and Symbolic Computation: An International Journal, 4, 3, 1991. This is a substantial revision of the originally published in OOPSLA ‘87.

[Wieringa94] R. Wieringa, W. de Jonge, P. Spruit, Roles and dynamic subclasses: a modal logic approach. In ECOOP´94 Proceedings, Springer-Verlag, 1994.