Omicron, An example of an Object-Oriented Calculus[1]

by Else K. Nordhagen (lc@ifi.uio.no, http://www.ifi.uio.no/~lc)
Department of Informatics, University of Oslo
P.O.Box 1080, Blindern, 0316 Oslo, Norway
phone: + 47 22 85 27 13
fax: + 47 22 85 24 01

Abstract:

Object-oriented technology featuring the concepts of object identity, instance creation, encapsulation, dynamic binding and inheritance are increasingly popular when designing systems, but there is a lack of formalisation tools. The common way to formalise object-oriented models is by translating from object-oriented models to models which are supported by existing calculi, ie, functional models (lambda calculus) or models of communicating processes (eg, π-calculus). Such translations are not ideal as it may not be intuitive how definitions and proofs done in these calculi map to object-oriented models. This paper presents a calculus which is based on the object-oriented concepts. Reasoning about object-oriented systems can then be done without changing paradigm.

Key words: Formal theory, object-oriented models, distributed systems

1 Introduction

Central concepts in the object-oriented tradition of, eg, (Robson 1981), (Stroustrup 1987), are object identity, instance creation, encapsulation, messages and dynamic binding and inheritance, which is something different from subtypes (America 1987). In the object-oriented tradition subtype relations follow the principle of substitutability expressed in (Wegner and Zdonick 1988) :

An instance of a subtype can always be used in any context which an instance of a supertype was expected.

The subtype relation used in the object-oriented design tradition focuses on the collaborative behaviour of objects, ie, how objects send messages to each other, change shared variables and create new objects in response to receiving messages. The Model-View-Controller design (MVC, see eg, (Gamma et al. 1994)) is a typical example of an object-oriented design. Other examples can for example be found in (Gamma et al. 1994) and (Jacobson et al. 1995).

To prove properties of objects which are relevant for this kind of behavioural subtype relation, it is necessary to be able to express object behaviours as described in object-oriented designs. The selected approach is creating a calculus which models such behaviour. The language resembles a mixture of SELF and Beta, but simplified to be practical as a calculus language and where objects execute in parallel. This has the advantage of being familiar to practitioners of object-oriented system development. Such a calculus and results from using the calculus, may therefore be easier to understand than using more unfamiliar modelling paradigms.

One important motivation for creating a formal framework is that the framework can be used in the search for necessary and sufficient requirements on object types to assure substitutability, ie, creating a type theory for collaborating objects. In order to find only necessary requirements it is important that no requirements are taken for granted. The specification and implementation language should therefore be defined with as few concepts as possible. Also, traditional object-oriented typing and other such ideas which are introduced to help developers write correct code should be avoided as such ideas have strong relationships to requirement assumptions. Section 2 describes a minimal language retaining the basic object-oriented design concepts.

---

[1] Accepted for NIK'96 - Norwegian Informatics conference 1996. Easiest available at http://www.ifi.uio.no/~lc/lit-LC.html

Section 3 presents the Omicron calculus which is based on the object-oriented concepts and the object behaviours: message sending to objects, object creation and updating of shared variables. The Omicron language resembles a mixture of SELF and Beta, but simplified to be practical as a calculus language and where objects execute in parallel. Section 4 gives a short presentation of how the Omicron calculus has been used for defining behavioural types for objects. This section also presents some conclusions from using the calculus.

Related work:
Functional models are often created for specifying and reasoning about objects. These approaches model objects as collections of functions, and object behaviour as function evaluations. This is not adequate for describing objects of the object-oriented tradition (Wegner 1994).

Process calculi such as the π-calculus focuses on how processes choose to send and receive signals on communication channels. Object-oriented concepts are not supported. However, it is possible to translate from objects to process descriptions, see eg, (Walker 1991). From such translations it is evident that the two paradigms are quite different and it is not intuitive how definitions and proofs done in, eg, the π-calculus translates to an object-oriented setting.

Some object calculi have been proposed. Several of these, for instance (Abadi and Cardelli 1994), focus on values and value transformations (functions) in objects rather than the object-oriented concepts. These have the same weaknesses as traditional functional models in relation to modelling object behaviour. Other object calculi such as (Nierstrasz 1993), have languages more related to languages for describing communicating processes such as the π-calculus. They only express message sending behaviour and no object creation actions. Also, they have synchronous message passing where as common object-oriented models have objects which are willing to receive any message. This corresponds with asynchronous message passing.

The Actor model (Agha 1986) is different to the present formalism in that their model of internal concurrency is server driven in that an object must explicitly create new internal threads, while the present calculus model client driven internal concurrency where new threads are implicitly created upon receipt of a message. The latter is common in most object-oriented programming languages and design notations.

C. B. Jones has developed a design notation named $\pi o \beta \lambda$ and used it to reason about object-based programs (Jones 1995). Jones has chosen to focus on a different set of object-oriented concepts than the present notation. For example $\pi o \beta \lambda$ has classes for object creation and each object can only have one method executing at the time, while the present notation uses object as templates for object creation and any number of methods can execute at the same time. The main difference is, however, that $\pi o \beta \lambda$ is typically used to reason about the state of shared variables while Omicron is created for reasoning about objects' reactive behaviour such as message passing and object creation.

2 Simplifying the Language

This section describes a simple object-oriented language created for describing object-oriented designs. The object-oriented language is simple as it has a very limited set of constructs. Ungar et. al. gives a good overview of some other language features which may be simulated by constructs in simple prototype-based languages (Ungar et al. 1991). The simple language presented here is named Omicron, after the short o of the Greek alphabet.

Omicron is not meant as a practically usable object-oriented programming language in itself, as many features which usually are handled by runtime systems have to be handled manually in Omicron. When developing object-oriented systems, the specifications, designs and implementations may be done using more user friendly notations such as an object-oriented programming language or design notations. Reasoning can then be done by translating from the more user friendly form to Omicron where proofs of, eg, refinement relations between specification and realisations can be done.

Advantages of limiting the number of concepts

One reason for creating Omicron is to understand more about categorisation (*typing*) of objects by studying congruence relations between objects and configurations of objects. Therefore the Omicron language should have as few kinds of elements as possible. The formal language will therefore have no distinction between type, class, object and method.

Another advantage of limiting concepts is that it reduces the number of concepts which must be defined, eg, homogenising objects and methods eliminates the need for a special syntax for distinguishing between them. Also, homogenising objects and classes by using objects as templates for other objects and allowing inheritance between objects eliminates the need for special syntax to distinguish classes and objects, but still retaining inheritance and template properties.

In traditional object-oriented systems objects are both described by sentences in a program and represented by bits in memory when the program is executed. This distinction between the program-description of an object and the representation of an executing object is eliminated in Omicron. This is done by having the language expressions represent both the program-description and the executable representation of an object.

This section elaborates on how concepts are homogenised and how this is done in a reasonable way.

Omicron objects have state and actions

This subsection presents some examples illustrating how objects are described and how sentences are executed.

In Omicron an object may be defined as in the following example:

e : ([s→o, w→m, t→j], s!w(t); s:=t; s := t clone;)

This defines an object named e with three "slots" s, w and t. The value of the s slot is the name o, the value of slot w is m and the value of t is j. The defined object also has a body consisting of three *sentences*:

a message send sentence	s!w(t)
an assignment sentence	s := t
a clone sentence	s := t clone

These sentences define operations the object performs in sequence when it executes. An executing object has an execution mark, denoted $, which indicate where the control is. When a sentence is executed the execution mark is advanced to the right of the executed sentence.

After execution of the first sentence in the object defined above, the object is described as follows:

e : ([s→o, w→m, t→j], s!w(t); $ s:=t; s := t clone;)

The execution of the object will continue with the sentence to the right of the execution mark. In the above case this means that the next sentence to execute is s:=t.

The result of executing the message send sentence is that the object named o gets the message m with parameter j. The result of executing the assignment sentence is that the slot named s gets the value j. The result of the clone sentence is that the object named j is copied, given a new unique name within the system, eg, k. The s-slot gets the new name as its value, in this case k.

The result of executing a sentence is described by an *action*. The actions from execution of the example object above are:

e->o!m(j)	a *message send action* from a sentence in the object named e: o gets the message m with j as a parameter
e->o.s:=j	an *assignment action* from a sentence in the object named e: the slot s in the object o gets the value j
e->o.s:=k/j	a *clone action* from a sentence in the object named e: the object j is copied and given a new name k and the slot s in the object o gets the value k

Omicron also includes a simple *if-sentence*: s:=(v=w t f)
This reads: if the v and w slots hold the same name then s gets the name in t, if not, s gets the name in f.
An example of an object with an if-sentence: e : ([s→j, v→k, w→k, t→i, f→j], $ s:=(v=w t f))
This will lead to the action: e->e.s:=i
In the case the object is defined as: e : ([s→j, v→k, w→l, t→i, f→j], $ s:=(v=w t f))

the action will be: e->e.s:=j

The assignment sentence s:=t may therefore be seen as a special case of the if-sentence: s := (t=t t t). Note that the names in the sentences are always slot names and never constants. This is done in order to be able to model message selectors as slot values, something which is found in, eg, Smalltalk (Goldberg and Robson 1983). The use of only slot names do not limit the language since a slot holding the desired constant value can always be defined and the name of this slot can always be used instead of the constant value.

Objects as templates for object creation

There are mainly two alternative ways of creating an object; one is to define the object explicitly and the other is to refer to a template to use for its creation. Both alternatives are available in Omicron.

An object may be explicitly defined as in the examples above. An existing object may also be used as a template, usually denoted prototype, for the creation of other objects. Creating an object by copying a prototype object is usually called cloning the prototype.

By functioning as prototype objects, Omicron objects also function as classes do in languages like C++ (Stroustrup 1986) and Smalltalk in that they are templates for object creation. This limits the number of concepts but does not limit expressability.

In several object-oriented languages it is possible to send templates for object creation as parameters in a message. This is possible in languages like, eg, Smalltalk and SELF. In Smalltalk classes are templates as well as objects, and may therefore be passed as parameters. In SELF, as in Omicron, objects are templates for other objects.

Inheritance between objects: Extension objects

By using extension objects a system can be designed so that certain objects can used to represent classes and extension objects represent subclasses. Extension objects are defined in (Blair et al. 1991) as follows:

> *Extension objects : In class based systems, a subclass defines some specialised structure and behaviour and inherits default structure and behaviour from its superclasses. How are such elements shared in classless systems ? Similarly, an extension object can be created that can share with one or more original prototypes. These objects, declared as shared from the view point of the extension objects, act as surrogates or proxies to which the extension objects will turn for assistance. More specifically, extension objects do not just share behaviour; they can share knowledge contained in their prototypes in a more general way; that is, the value of state variables can be shared between objects.*

In (Blair et al. 1991) the behaviour is not seen as the sentences in the body of an object, but rather as the set of methods available to the object. Extension objects directly correspond to Omicron objects with inheritance. Omicron objects can inherit slots from each other and slots are used both as traditional variables *and* for holding methods. Note that only slots are inherited, not sentences.

To specify inheritance a special kind of slot is introduced: an *inheritance slot*. In SELF an inheritance slot is specified by a trailing star (in SELF syntax: slotName*). This is adopted in Omicron using slotName*.

An example of a configuration of two objects with an inheritance slot:

p : ([x→o, w→m, t→j],)
e : ([h* →p], x!w(t); x:=t; x := t clone;)

where h is the name of the inheritance slot. The object named e will then inherit all slots of the object named p. e therefore inherits the slots x, w and t and the sentences in e refer to these slots in p.

There may be more than one inheritance slot in an object and the value of the inheritances slots may be changed just like any other slot's value. An example of how to make an exact definition of inheritance between objects is given in section 3.2.

Objects and methods in one concept

In Omicron methods are objects which are extensions of other objects, ie, they inherit from the object they operate on. Method objects are described by (Blair et al. 1991) as follows:

> *Method objects - Everything in this type of object system is an object. Thus state and interface methods can both be considered objects in their own right. Consider methods first. The messages that an object receives are handled by a set of methods. Each method has a name and a response. Thus on receipt of a message, an object sends a delegate message to each of its methods. This message carries the name of the client (i.e. the original sender of the delegate message) as an argument. … Each method object checks that the type of the message matches its own name. If identical, the method object accepts the message and invokes its response.*

In Omicron a method is not executed by sending it a message in order to evoke its response. The execution is assured by what corresponds to a runtime system, which select a method. A method is selected by checking that the selector of the message matches a name of some slot in the receiving object. If a match is found then the object referenced in the matching slot will be executed giving the methods response.

Methods have in-parameters. For this purpose the Omicron language include syntax for defining *input-slots*. Such input-slots are defined by starting the slot name with colon (:), eg, :slotName as in SELF.

An example showing how messages are sent and received:
We have three objects, a sender, a receiver and a method defined as follows:

sender	: ([x→receiver, w→m], $x!w(x);)
receiver	: ([m→method, y→o, t→j],)
method	: ([:s✯→nil], y:=t;)

When the sentence in the sender is executed a message is sent to the receiver. The action will be:

sender->receiver!m(receiver)	meaning the receiver gets the message m with the object name 'receiver' as parameter

When the message m is sent to the receiver, the underlying execution mechanism looks for a slot named m. In this case the receiver has an m-slot with value 'method'. To model the execution of a method the method object is then copied and a $ inserted at the beginning of the sentences in the method copy:

method-copy : ([:s✯→receiver], $y:=t;)

If there were execution marks in the original method, then these are removed so that there will only be one execution mark at the time in an object. In the method-copy the value of the input-slot named s has been updated to be equal to the parameter to the message, here 'receiver'. In addition to being an input-slot, the s-slot is an inheritance slot. This combination of input and inheritance slot models how a method executes on behalf of the object who received the message, ie, is an *extension object* to the receiver. This s-slot corresponds with the pseudo-variable 'self' in Smalltalk, the implicit 'self' in SELF and 'this' in C++ and Simula.

The main difference between Omicron and the other mentioned languages on the matter of the 'self/this'-variable is that slots used as self/this must be explicitly defined as a parameter in Omicron methods. This is done automatically in the other languages.

As the message send sentence in the sender is executed the execution mark is moved to the right, ie, the sender becomes a terminated object which has no more sentences to execute:

sender : ([x→receiver, w→m], x!w(x); $)

After the message has been sent and the method copied, the configuration of objects looks like:

sender	: ([x→receiver, w→m], x!w(x); $)
receiver	: ([m→method, y→o, t→j],)
method	: ([:s✯→nil], y:=t;)
method-copy	: ([:s✯→receiver], $y:=t;)

where the sender has terminated since there are no more sentences to execute and the method-copy is ready to execute its assignment sentence since this is preceded by $. Making method-copies accommodate recursion. This gives expression power which includes while-constructions.

As mentioned above, the method-copy executes on behalf of the receiver. This means that the method-copy has access to the slots in the receiver so that these slots may be read and set by operations in the method-copy. This access is given by the definition of the s slot which is both an input and inheritance slot. Therefore, after execution of y:=t in the method-copy the configuration of objects will look like:

sender	: ([x→receiver, w→m], x!w(x); $)
receiver	: ([m→method, y→j, t→j],)
method	: ([:s☆→nil], y:=t;)
method-copy	: ([:s☆→receiver], y := t; $)

There are now two terminated objects and the slot y has the value j.

Method-copies must be extension objects
The method-copy objects are extensions to other objects in that they manipulate the other objects state. If the method-copies are not extension objects then there is no way to influence an object's state by sending it a message. This is because sending a message can not result in the execution of operations which change the values of the object since the non-extension method-copy is not allowed to access the object's slots.

3 Omicron Syntax and Semantics

3.1 Omicron syntax

This section describes the syntax of the Omicron language through the use of extended BNF. Terminal symbols are given in **bold font**.

Configuration	::=	$Object^{*}_{\|\|}$	
Object	::=	name : **(**Slots**,** Body**)**	-- Definition of an object
Slots	::=	**[** $SlotDef^{*}_{,}$ **]**	
Body	::=	$Sentence^{*}_{;}$ | $Sentence^{*}_{;}$ **$** $Sentence^{*}_{;}$	
Sentence	::=	name **:=** Ref **clone** |	-- Clone sentence
		$name^{+}_{,}$ **:= (**Ref = Ref Ref Ref **)** |	-- If-sentence
		Ref **!** Ref **(** $Ref^{*}_{,}$ **)**	-- Message send sentence
SlotDef	::=	slotName →name	
Ref	::=	name | **this**	
slotName	::=	name |	-- Plain slot
		:name |	-- Input slot
		name☆ |	-- Inheritance slot
		:name☆	-- Input and Inheritance slot
name	::=	$char^{+}$ but no colon (:) first and no ☆ last and not equal to 'this'	
char	::=	**a** | … | **z** | **A** | … | **Z** | **0** | … | **9** | **+** | **-** | ***** | **/** | **_** | **:** | **/** | **=** | **_**	

where $Object^{*}_{||}$ means $Object_1$ || … || $Object_n$ and $name^{+}_{,}$ means one or more names separated by comma. The symbol $ is called the execution mark. As objects may be executing in parallel, there may be more than one execution mark in the system, but only one in each object to keep the system well defined. The transition rules of section 3.2 define how execution marks are handled. **this** is a special name which is an alias for the name of the object where **this** is found. **this** may be viewed as a relative object name representing the object name of the surrounding object.

Note that there may be three types of slots: plain slots, input slots and inheritance slots. A slot may be both an input slot and an inheritance slot. The plain slots have no other purpose than to store names. Input slots are slots for storing input names (parameters) when the object is executed as result of some object receiving some message. The third type of slots are inheritance slots.

A *syntactically correct* configuration is a configuration which could be derived by using the syntax rules and where each object has a unique name. An *system* is a configuration which is closed in that the objects in the system only collaborate with each other.

3.2 Formalisation of configurations

The set of all configurations is denoted $\mathscr{C}$. The set of all Slots is denoted $\mathscr{M}$, the set of all names is denoted $\mathscr{N}$. The following syntactic conventions are used: e,f,i,j,o,p denote object names, k,l denote names used as new object names, v,w,s,t,u denote slot names, m denotes a name used as message selector, $\tilde{v}$ denotes a list of input slot names, $\tilde{p}$ denotes a list of object names used as parameters (slot values), A,B,C,D denote configurations of objects, M denotes a Slot map and S denotes an object body, ie, a sequence of sentences.

Inheritance:
If an object has an inheritance slot it will inherit slots of the object referred to in the inheritance slot. To inherit slots means that sentences in the body of the object may use the names of the slots in the inherited slot map. An object may have several inheritance slots in its slot map, and also an inherited slot map may contain inheritance slots. One object's total slots is therefore all the slots found in the slot maps in the map graph defined by the values in the inheritance slots of the maps. This is referred to as an object's *inheritance graph*.

When looking up a given slot name in a Slot map the search is always started in the Slot map of the object where the slot name is referenced. E.g. in the configuration

a: ([s→a],) || b: ([$t^{\star}$ → a, w→m], s!w())

the lookup of the slot x referred to in the sentence in d is started in d's Slot map. If the slot name is not found in the first Slot map the search is continued at the next level in the graph, ie, in a's map. In general there are many alternative ways to traverse an inheritance graph and some versions are discussed in (Chambers et al. 1991). We call the object where a particular slot is found for the *owner of the slot* , and denote the owner of the s slot as seen from the object named o defined in the configuration C for owner(C, o, s). owner(C, o, **this**) is o.

In Omicron the inheritance graph of an object may be changed during execution as inheritance slots may get new values just like all other slots. An inheritance slot may also be an input-slot which then is defined by the syntax: :slotName*.

The following sequence notation is used in the below: $\langle a_1, \ldots, a_n \rangle$ denote a sequence of the n items a_1 to a_n, # is a function which value is the length of a sequence. The following notation is used to describe various aspects of objects and configurations:

o ∈ D	== o ∈ D.Dom
C(o).Body	== the Body of the object named o
C(o).Slots(s)	== the value of the slot s in the Slots of the object named o This is referred to as *getting the value* of the slot s in the object o. If o is not in C.Dom or s is not found in Slots, the result is the object name nil. Note that C(o).Slots(**this**) will always give o.
C(o).inputs	== the sequence input slot names in the Slot map of the object named o
@C(o:s)	== Returns true if there is a slot named s in the slot map of an object in the inheritance graph of the object names o, false otherwise. When this function return true it is said that *the slot s has an owner*.
C(o:s)	== C(owner(C, o, s)).Slots(s)
C[o.s:=j]	== *if* @C(o).Slots(s) *then* C[o→(C(o).Slots[s→j], C(o).Body)] *else* C. This is denoted *setting the value* of the slot named s in the object named o to the value j. If o is not in C.Dom or s is not found in Slots, the result is no change to C.
C[o:s:=j]	== C[owner(C, o, s).s:=j]
C(o:$\langle s_1,\ldots,s_n \rangle$)	== $\langle C(o:s_1),\ldots,C(o:s_n) \rangle$. Similar for other uses of sequences of slot names
$\tilde{o}.\tilde{s}$	== $o_1.s_1, \ldots, o_n.s_n$ where $\tilde{o} = o_1, \ldots, o_n$ and $\tilde{s} = s_1, \ldots, s_n$

3.3 Formal operational semantics of sentences

There are basically two different views on sentences. They may be seen as atomic operations to be performed, or as expressions which are to be evaluated. The first alternative is found in process languages like CCS and the π-calculus while the latter is found in functional languages such as ML (Milner et al. 1990). The object paradigm has relations to both of the above traditions. When defining the language for describing objects, one or the other alternative should be selected to keep the syntax and semantics simple. Different alternatives have been tried, eg, a version with expression evaluation semantics is found in (Nordhagen 1997). Here the conclusion was that atomic operation semantics gave the simplest calculus. Therefore, in Omicron sentences are seen as definitions of atomic operations.

The formal semantics of Omicron sentences is defined by a set of transition rules creating an operational semantic definition following the tradition started by (Plotkin 1981). The syntax of the transitions is given below together with an informal description of the semantics.

Definition: Transition and action
A transition is of the form:

$$A \xrightarrow{\alpha} A'$$

Intuitively, this transition means that the configuration A can evolve into A', and in doing so perform the action α. The set of all such actions is denoted $\mathscr{A}$. In Omicron there are four types of actions a configuration can perform. The actions are first described formally through a transition system.

The transitions may be applied to a configuration in any order, as long as the transition is legal by the premises in the rules of action. The rules of action are not confluent, ie, the result configuration after applying a set of transitions to a configuration may depend on the order in which the transitions are applied. This is in correspondence with the intuition that the result of executing a program depends on the execution order.

Definition: Transition relation and rules of action
The transition relation, denoted $\xrightarrow{\alpha}$, is the smallest relation between object configuration expressions satisfying the following *rules of action*. All the names in the rules are arbitrary and may be replaced with any other names. Informal descriptions of the rules are given after the list of rules. By definition || is associative and commutative.

The rules are given by transition for the three different kinds of sentences defined for Omicron: if-sentence ($\tilde{s}$:=(v=w t u)), clone sentence (s := t clone) and message send sentence (s!w($\tilde{t}$)). We are here considering a system C of the form C == C'||e:(M, S_1 $ sentence; S_2) where S_1 and S_2 are sequences of sentences, possibly empty.

IF rules :
IF-true rule:

$$\frac{@C(e :< \tilde{s}, v, w, t >) \ \wedge \ C(e : v) = C(e : w)}{C' \parallel e : (M,\ S_1 \$\tilde{s} := (v = w\ t\ u); S_2) \xrightarrow{e->\tilde{o}.\tilde{s}:=j} (C' \parallel e : (M,\ S_1; \tilde{s} := (v = w\ t\ u); \$S_2))[e : \tilde{s} := j]}$$

where C(e:t)=j and $\tilde{o}$ = owner(C,e, $\tilde{s}$)

IF-false rule:

$$\frac{@C(e :< \tilde{s}, v, w, u >) \ \wedge \ C(e : v) \neq C(e : w)}{C' \parallel e : (M,\ S_1 \$\tilde{s} := (v = w\ t\ u); S_2) \xrightarrow{e->\tilde{o}.\tilde{s}:=j} (C' \parallel e : (M,\ S_1; \tilde{s} := (v = w\ t\ u); \$S_2))[e : \tilde{s} := j]}$$

where C(e:u)=j and $\tilde{o}$ = owner(C,e, $\tilde{s}$)

CLONE rule:

$$\frac{@C(e : s) \ \wedge \ C(e : t) = j \in C}{C' \parallel e : (M,\ S_1 \$s := t\ \text{clone}; S_2) \xrightarrow{e->o.s:=k/j} (C' \parallel e : (M,\ S_1; s := t\ \text{clone};\ \$S_2) \parallel k : C(j))[e : s := k]}$$

where j = C(e:t), o = owner(C,e,s) and $k \notin C$

SEND rule:

$$\frac{@C(e:\tilde{t}) \ \wedge \ C(e:s) \in C \ \wedge \ C(o:m) \in C \ \wedge \ \#C(j).inputs \ = \ \#\tilde{t}}{C' \parallel e:(M,\ S_1\$s!w(\tilde{t});S_2) \xrightarrow{e->o!m(\tilde{p})/k} C' \parallel e:(M,\ S_1 s!w(\tilde{t});\$S_2) \parallel k:(C(j).Slots[\tilde{v} \rightarrow \tilde{p}],\ \$C(j).Body^{-\$})}$$

where o = C(e:s), m = C(e:w), j = C(o:m), $\tilde{v}$ = C(j).inputs, $\tilde{p}$ = C(e:<$\tilde{t}$>), k $\notin$ C and

$C(j).Body^{-\$}$ means that any existing $ in the body is removed

ERROR rule:

$$\frac{\text{no other rule of action is applicable to the e - object}}{C' \parallel e:(M\ S_1\ \$\ \text{sentence};\ S_2) \xrightarrow{e->error} C'}$$

Explanations of the rules are given in (Nordhagen 1997). Here is just given an explanation of the SEND rule and a comment on the error rule:

Execution of a message send sentence gives a *message-send* action of the form e->o!m(i_1,…,i_n)/k, meaning that the actions stem from execution of a sentence in an object names e and o gets the message m(i_1,…,i_n) and k is the name of a new object created as result of the message send.

In an action e->o!m(i_1,…,i_n)/k, o is called the *receiver* of the message, m is called the *selector* of the message and $\tilde{p}$ is a list of names called the *parameters* of the message. The object named j in the SEND rule where C(o:m) = j is called a *method*.

The rule is applicable when there is an execution mark in front of a message send sentence, all the slots have owners and both a receiver object and a method object is found. It is also required that the m-slot of the receiver (o) must hold the name of an object in the configuration with the same number of input slots as there are parameters in the message.

The rule models the reception of a message as follows: The method is copied, resulting in a *method copy* which is given a new name (k) and its input slots (C[k].inputs = $\tilde{v}$) are updated with the message's parameters $\tilde{p}$ (ie, ($\tilde{v} \rightarrow \tilde{p}$)). Also, any old execution marks are removed and a new execution mark, $, is inserted into the body of the method-copy.

Note that the rules of action preserves syntactic correctness since applying a rule of action to a syntactic correct configuration gives a syntactic correct configuration.

The ERROR rule is applicable when there is an execution mark in front of a sentence and no other rules apply to this sentence. One reason for an error is that there are slots which do not have owners. Another reason is that there are slots which should have values which are names of objects in the configuration, but which are not. This would create "strange" actions such as messages to non-existent object or cloning of non-existent object. The error rule says that the object holding the erroneous sentence is removed from the configuration. This reflect a system-view where an object with an erroneous sentence terminates. Each object is seen as executing separately from the rest of the system in that only the object with the erroneous sentence terminates and the rest of the system continue execution.

Observation I : The rules of action preserves syntactic correctness
Applying a rule of action to a syntactic correct configuration gives a syntactic correct configuration.
Proof: Done by cases for the different rules. ❑

Definition: Terminal configurations
A configuration is terminal if all object bodies either has an execution mark at the end of its sentences or has no execution mark. The set of all terminal configurations is denoted $\mathscr{C}_{\text{Term}}$.

$$\mathscr{C}_{\text{Term}} = \{\ C \in \mathscr{C} \mid \ \forall\ o \in\ C : \$ \notin C(o).Body \vee (\exists\ S : C(o).Body = S\$)$$

Definition: Actions

$$\forall\ C \in \mathscr{C}\text{: actions}(C) = \ \{\ \alpha \mid \exists\ C' :\ C \xrightarrow{\alpha} C'\ \}$$

Proposition 1 No rules of action are applicable to a terminal configuration

$C \in \mathscr{C}_{Term} \Rightarrow$ actions(C) = Ø

Proof: No rules are applicable to (M, S $) or when $ is not in the body of the object. ❑

Observation II : Either a configuration is terminal or rules of action are applicable

Either a configuration C is a terminal configuration or at least one of the rules of actions are applicable:

$C \in \mathscr{C} \Rightarrow C \in \mathscr{C}_{Term} \vee \exists \alpha \in$ actions(C)

Because:

If $C \in \mathscr{C}_{Term}$ then no rules of action are applicable by proposition 1. If $C \notin \mathscr{C}_{Term}$ then there exists one or more objects with an execution marks in front of a list of sentences. It must therefore exist one or more of the following alternative object definitions in the configuration:

Alternative	*Applicable rules*
(M, S_1 $ s := t clone; S_2)	The CLONE rule or the ERROR rule is applicable.
(M, S_1 $ $s_1 \ldots s_n$:= (v=w t u); S_2)	The one of the IF rules or the ERROR rule is applicable.
(M, S_1 $ s!w($t_1 \ldots t_n$); S_2)	The SEND rule or the ERROR rule is applicable.

❑

4 Conclusion and further work

Omicron concepts map directly to object-oriented concepts: object identity, instance variables, methods, object creation, encapsulation, inheritance and messages are sent to objects, as opposed to functional concepts of values and function applications and process models of signals on channels and spawning of processes. Expressions and reasoning done using Omicron may therefore be more intuitive to object-oriented designers and programmers than the other existing calculi as there are more commonalties between Omicron and object-oriented languages than between these languages and the concepts in the other calculi.

Omicron enables reasoning about the compositionality of configurations of objects based on the observable behaviour of objects. The calculus can therefore be used to study object types in relation to compositionality and substitutability of software components which are specified by the observable behaviour of the components. Some work have already been done in this direction in (Nordhagen 1997). Space do not permit a detailed presentation of this work. A very short summary is given below. In (Nordhagen 1997) the following concepts are defined:

Definition: Sequences of transitions and actions: $\xrightarrow{\bar{\alpha}}$, $\bar{\alpha}$

$C \xrightarrow{\bar{\alpha}} C'$ denotes any finite sequence of zero or more transitions $\xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n}$ from C to C' as defined by the rules of action and $\alpha_i \in \mathscr{A}$. A sequence $\xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n}$ ($= \xrightarrow{\bar{\alpha}}$) may also be written as the list of actions $<\alpha_1,\ldots,\alpha_n>$ ($= \bar{\alpha}$).

Definition: The traces of the executions of a configuration

The traces of the executions of a configuration C are all the sequences of actions leading up to a terminal configuration derived from C:

$$Traces(C) = \{ \bar{\alpha} \mid \exists C' \in \mathscr{C}_{Term} : C \xrightarrow{\bar{\alpha}} C' \}$$

Definition: Observable Actions

Given an action α and a configuration of objects D. D is typically a part of a system, not a full system in itself. The action α is observed by the configuration C if the action change slots in an object in D, clone an object in D, is a message send to an object in D or an error action from an object in D. The actions which are observable by the objects in D is denoted obs(D) and is formally defined:

$$\begin{aligned} obs(D) = \{ & \ e\text{->}o_1.s_1,\ldots,o_n.s_n := j && \mid \exists i \in \{1\ldots n\} : o_i \in D \} \cup \\ & \{ \ e\text{->}o.s := k/j && \mid o \in D \vee j \in D \} \cup \\ & \{ \ e\text{->}o!m(\tilde{p})/k && \mid o \in D \} \cup \\ & \{ \ e\text{->}error && \mid e \in D \} \end{aligned}$$

"α is observable by D" means $\alpha \in$ obs(D).

Definition: Observed Trace

Given configuration B and B. The D-observable trace of the system B||D are the actions in the trance of B||D which are in obs(D). The D-observable trace of B||D is denoted $Trace_D(B||D)$

Behavioural equivalence classes for objects can be defined by using the above definitions. All objects of the same behavioural equivalence class as a configuration B relative to an observing configuration D are related by a refinement relation which is defined as follows:

Definition: Refinement relation

Given configurations $A||D \in \mathscr{S}$ and $B||D \in \mathscr{S}$. A is a refinement of B relative to D, denoted $A \leq_D B$, if:

for every sequence of D-observable actions from A||D
then there is an equal D-observable sequence of actions from B||D.

This can be formally defined as follows:

$$A \leq_D B \Leftrightarrow \forall\ \overline{\alpha} \in Traces_D(A||D)\ \exists \overline{\beta} \in Traces_D(B||D) :\ \overline{\alpha} = \overline{\beta}$$

This definition is a simplified version of the one given in (Nordhagen 1997), as the complete definition includes details on how to handle differences in object names in the configurations A and B, while still defining the actions in the observable traces as observably equal.

Based on the definition of the refinement relation, a theorem showing substitutability properties is proven. The theorem can be stated in a simplified form as follows:

$$A \leq_D B \wedge C \leq_B D \Rightarrow A \leq_C B \wedge C \leq_A D$$

The idea behind this proposition is that there is a system B||D where B and D denote configurations of objects. Both B and D may be replaced by new objects as long as the new objects have the same observable behaviour as the old ones. Ideally it should be possible to separately develop the new configurations of objects, above denoted A and C. The proposition express this idea. It says that if A and C are separately tested and A is found D-observably equal to B and C is found B-observably equal to D, then A and C will also be observably equal to B and D respectively as observed by C and A. This assures that no new functionality and/or untested behaviour will occur in the new configuration A||C as compared to the behaviour observed when executing A||D and C||B.

The above proposition can be generalised to express the same properties for a system with more than two components. This more general proposition is defined as follows:

Definition: Reliable refinements and the substitution proposition

A refinement relation, $\leq$, is said to be a *reliable refinement relation* if the following *substitution proposition* can be proven:

$$(\forall i \in \{1..n\} : C_i \leq_{D\text{-}i} D_i) \Rightarrow (\forall i \in \{1..n\} : C_i \leq_{C\text{-}i} D_i)$$

where D_i denote the i'th component of D and C_i denote the i'th component of C and D-i denote all Ds except D_i and C-i denote all Cs except C_i

In relation to the substitution proposition D may be seen as a specification of the n components of some system, where the n components are denoted $D_1 \ldots D_n$. The new version of these n components are denoted $C_1 \ldots C_n$.

The term *safe substitution* is introduced to express the above formally defined properties:

Definition: Safe substitution

If a component C is a reliable refinement of a component D relative to a context B then we say that C may safely substitute C in B or any context which is a reliable refinement of B.

The work in (Nordhagen 1997) proves the substitution proposition. The work shows that to prove the proposition and assure safe substitution it is necessary to define a set of requirements on object descriptions and congruence relations. These requirements correspond with object-oriented language design choices, eg, Simula (Dahl et al. 1968), Smalltalk and Dylan (Apple 1992). This supports the idea that Omicron formalise object-oriented concepts as found in existing object-

oriented languages while minimising the number of concepts. It is clear that this minimisation is done at the expense of properties which help give safe substitution such as static inheritance structures, static class definitions and encapsulation of variables.

The rules also correspond with experienced object-oriented designers' rules for making reusable software components as expressed in (Johnson and Foot 1988), (Gamma et al. 1994) and others. For example there are rules which argue that class names in the code reduce reusability. No other formal work has so clearly found proofs of the necessity to follow such reuse-rules. This suggests that the above definitions of safe substitution, refinements and observable behaviour is more in accordance with object-oriented design practices than other congruence definitions based on other formalisms.

Acknowledgements
Olaf Owe and Bjørn Kirkerud have given useful comments on earlier version of this papers. The research is supported by scholarships from the Norwegian Research Foundation and the University of Oslo, Department of Informatics.

5 References

Abadi, Martín and Cardelli, Luca, (1994) *A theory of primitive objects*, Digital Equipment Corporation, DEC-SRC, Shorter versions in European Symposium om Programming, and in Theoretical Aspects of Computer Science 1994

Agha, Gul, (1986) *A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass.

America, Pierre, (1987) *Inheritance and Subtyping in a Parallell Object-Oriented Language*, European Conference on Object-Oriented Systems 1987, published in LNCS 276, pp. 234-242.

Apple, (1992) *Dylan, an object-oriented dynamic language*, Apple Computer, 1 Main Street, Cambridge, MA 02142, To order: email: dylan-manual-request@cambridge.apple.com

Blair, Gordon, Gallagher, John, Hutchison, David and Shephard, Dough, (1991) *Object-Oriented Languages, Systems and Applications*, Pitman Publishing, London, ISBN : 0-273-03132-5

Chambers, Craig, Ungar, David, Chang, Bay-Wei and Hölzle, Urs, (1991) *Parent and Shared Parts of Objects: Inheritance and Encapsulation in SELF*, in Lisp and Symbolic Computation 1991, 4:3

Dahl, Ole Johann, Myhrhaug, Bjørn and Nygaard, Kristen, (1968) *SIMULA 67 Common Base Language*, Norwegian Computing Center,

Gamma, Erich, Helm, Richard, Johnson, Ralph and Vlissides, John, (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 416, ISBN : 0-201-63361-2

Goldberg, Adele and Robson, Dave, (1983) *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass.

Jacobson, Ivar, Bylund, Stefan, Jonsson, Patric and Ehneboom, Staffan, (1995) *Using contracts and use cases to build pluggable architectures*, in JOOP May 1995, 8:2, pp. 18-76

Johnson, Ralph E. and Foot, Brian, (1988) *Designing Reusable Classes*, in JOOP 1988, 1:2, pp. 22-25

Jones, C. B., (1995) *Accomodating Interference in the Formal Design of Concurrent Object-Based Programs*, in FMiSD, Kluwer Academic Publishers, Boston. Manufactured in the Netherlands, 1:19

Milner, R., Tofte, M. and Harper, R., (1990) *The definition of Standard ML*, The MIT Press, ISBN : 0-262-12355-9

Nierstrasz, Oscar, (1993) *Composing Active Objects*, in Research Directions in Concurrent Object-Oriented Programming, ed: G. Aga, P. Wegner and A. Yonezawa, The MIT Press, Cambridge, Massachusetts, pp. 151-171, ISBN : 0-262-01139-5

Nordhagen, Else K., (1997) *A Formal Framework for Verification of Object Component Substitutability*, Doctoral Thesis, Department of Informatics, University of Oslo, Forthcomming spring 1997

Plotkin, G., (1981) *A structural approach to operational semantics*, Computer Science Department, Aarhus University, DAIMI FN-19

Robson, David, (1981) *Object-Oriented Software Systems*, in BYTE August 1981, pp. 74-86

Stroustrup, Bjarne, (1986) *The C++ Programming Language*, Addison Wesley

Stroustrup, Bjarne, (1987) *What is "Object-Oriented Programming" ?*, European Conference on Object-Oriented Programming 1987, pp. 57-76.

Ungar, David, Chambers, Craig, Chang, Bay-Wei and Hölzle, Urs, (1991) *Organising Programs Without Classes*, in Lisp and Symbolic Computation 1991, 4:3

Walker, David, (1991) *π-Calculus Semantics of Object-Oriented Programming Languages*, TACS 1991, published in LNCS 526, pp. 532-547.

Wegner, Peter, (1994) *Models and Paradigms of Interaction,* in Object-Based Distributed Programming, ECOOP'93 Workshop, Vol. 791, ed: R. Guerraoui, O. Nierstrasz and M. Riveill, Springer-Verlag, pp. 1-32, ISBN : 3-540-57932-X

Wegner, Peter and Zdonick, Stanley B., (1988) *Inheritance as an Incremental Modification Mechanism or What Like Is or Isn't Like*, European Conference on Object-Oriented Programming 1988, Oslo, Norway, published in LNCS 322, pp. 55-77.