Relations as Object Model Components

Jan Bosch*
Department of Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25, Ronneby, Sweden

Tel: +46-457-787 26
Fax: +46-457-271 25
E-mail: Jan.Bosch@ide.hk-r.se
URL: http://www.pt.hk-r.se/~bosch

Abstract

Although object-oriented methods make extensive use of relations between objects, these relations, other than *inheritance* and *part-of*, can not directly be represented by the conventional object-oriented model. This means that relations which are identified during analysis and design have to be implemented *on top of* the object model, i.e. by using method code and message passing, rather than by expressing relations directly within in the model. It would be beneficial if the object-oriented model would support the specification of all relevant types of relations within the model, including application-domain specific relation types. Therefore, we propose a mechanism, implemented in LAYOM – an extended object model, that supports specification of all types of relations between objects within the model as components of the object model. In addition, an approach for identifying and specifying application-domain relation types is presented.

Research Paper : language design, reuse, architecture

1 Introduction

The concept of relation plays an important role in modelling applications. Most object-oriented development methods, e.g. [8, 10, 16, 17], make extensive use of relations between analysis and design entities. However, the object-oriented model in which the analysis and design model generally is implemented does not provide support for expressing relations other than *inheritance* and *aggregation*. We consider this a problem because a relation, being a conceptual entity, can not be expressed in the object model as an entity, but needs to be implemented as distributed pieces of method code and messages. The software engineer should be able to express all types of relations, or at least the larger part, as entities in the object-oriented model.

Others [16] also identified this as a problem and propose to model relations as independent objects. This, we believe, is unsatisfactory for two reasons. First, a relation should be part of the object whose behaviour is influenced by the relation. A relation is not an object, but it should be modelled as an identifiable unit. Second, for the objects involved in the relation the designer still needs to define methods and message exchanges.

We propose to model the relations an object has with other objects as a part of the object definition. The designer can define relations of a class with other classes for three purposes. First, a relation can

*This work has been supported by the Blekinge Forskningsstiftelse.

be used to *extend* the functionality of a class. We refer to this type of relation as *structural relations*. Structural relations define the structure of a class and provide *reuse*. The second type of relation, i.e. a *behavioural relation*, is defined for clients of the class. The functionality of the class is used by client objects and the class can define a *behavioural relation* with each client (or client category). Behavioural relations *restrict* the behaviour of the class. For instance, access to the class might be restricted for the client in certain object states. The third type of relations that can be applied by the designer are *application domain relations*. Many domains have, next to reusable application domain classes, also application domain relation types that can be reused. In practice, we have experienced this categorisation as being very natural. Therefore, we categorise relations in *structural*, *behavioural* and *application-domain specific* relation types.

The contribution of this paper, we believe, is that it presents a novel approach to representing relations between objects that is 1) more uniform than the conventional approaches, 2) provides reuse of relation types and 3) increases the traceability of relations in the implementation model. We explicitly do *not* aim at modelling relations as in relational databases. Our goal is to extend the relation types that can directly be represented within the object model (currently *inheritance* and *aggregation*) with additional relation types that increase the expressiveness of the object-oriented model.

This paper is organised as follows. First, we analyse the problems of the conventional object model[1] with respect to relations in the next section. Then, in section 3, we introduce the process control domain which will serve as an example throughout the paper. In section 4, the aforementioned categories of relations are defined and described. As a solution to the discussed problems we propose the *layered object model* in section 5. Section 6 discusses the relation to other work and we conclude in the last section with an evaluation and a description of future work.

2 Rationale

In this section we describe the rationale for the research resulting in this paper. First, a short overview of object-oriented analysis and design is given. In the subsequent section, an analysis of the problems of the conventional object-oriented model with respect to the methodology are described.

2.1 Overview of Analysis and Design Steps

A conventional object-oriented method consists of many steps and processes, but it can generally be decomposed in five high-level steps. Below, we give a brief description of the steps. However, the intention of this description is to illustrate how a software engineer deals with relations during analysis and design. It explicitly does *not* intend to give a precise description of an analysis and design method. The description is a generalisation of the more popular object-oriented methods, e.g. [4, 8, 10, 16].

1. *Requirement and domain analysis*: During requirements analysis the software engineer tries to identify and define all requirements in a consistent, coherent and complete requirement specification. Domain analysis is concerned with identifying theoretical and engineering domains that are relevant for the application under development.

2. *Object and subsystem identification*: The software engineer structures the system into a hierarchy of subsystems and identifies relevant objects in the application domain.

3. *Relation identification*: After the system has been structured and candidate objects have been defined, the software engineer searches for relations between the identified objects. At this stage

[1] With 'conventional object model', we refer to the traditional object model as supported by Smalltalk and C++.

the types of the identified relations are unrestricted and the software engineer tries to express the characteristics of each relation as precisely as possible.

4. *Structure specification*: As the structure of the system is the focal point here, the relations identified during the previous phase are 'mapped' to one of three relation types, i.e. *inheritance*, *aggregation* or *uses*. The latter relation basically captures all relation types that cannot be expressed directly within the object model. The structure of the system is defined based on the inheritance and aggregate relations.

5. *Behaviour specification*: The *uses* relations, i.e. the relation types that cannot be implemented within the object model, are translated into method code and message sends between objects.

From the above description one can conclude that, although many different types of relations can be identified during the relation identification phase, only inheritance and aggregation relations are represented within the model. Other relation types have to be expressed *on top of* the object model. In addition, despite the fact that application domain entities are modelled by classes, the application domain relations between the classes are not modelled as first-class entities, but need to be translated into method code and message sends. Our experience has shown that, next to application-domain specific classes, often application-domain specific relation types also provide useful and reusable abstractions.

2.2 Problems

The problems related to the use of relations in the conventional object model can be categorised as follows.

- **Lack of Uniformity**: During analysis and design, the application is modelled in terms of objects[2] and relations between objects. During the implementation phase, the model has to be expressed in an object-oriented language that only supports the representation of the *inheritance* and *part-of* relation. Other types of relations have to be implemented on top of the language model. Obviously, this is not a uniform approach to expressing relations between objects: depending on the relation type, the software engineer has to express the relation in different ways in the object model.

- **Neglected Source of Reuse**: From our experience, we have learned that application domains not only have domain specific classes, but often also domain specific relations. However, because the conventional object model does not support explicit representation of relations, the domain specific relation types are generally neglected. A problem is that one looses an important source of reuse when relations are implemented as method code within one (or both) of the related objects. The first-class representation of the application domain relation type is lost and it cannot be reused.

- **Deminished Traceability**: If relations can not be expressed explicitly in the object model but have to be implemented as pieces of method code and messages between objects, the *traceability* of the implementation model with respect to the design model is more difficult. This because a unit in the design model does not have a corresponding unit in the implementation model. The implementation models the real world problem less exact than possible.

[2]When we use the term 'object' we refer, unless explicitly stated, to analysis and design entities, e.g. subsystems, classes and instances.

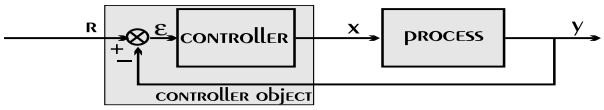

Figure 1: Default process - controller configuration

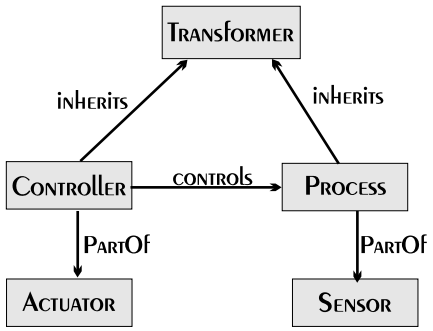

Figure 2: Analysis and design model

3 Example: Process Control Application

In this paper we will use a part of a process control application as the running example. In figure 1 the default configuration for a process and an associated controller is shown.

In process control, the system, i.e. the controlled and the controlling part, is build by using two types of components: *process* and *controller*. The process has one input, the *desired value* x, and one output, the *actual value* y. The controller has two inputs, the *desired value* r and the *measured value* y and one output, the *set value* x. The set value of the controller is connected to the desired value of the process.

In figure 2, an analysis and design model of a part of a process control application is shown. The most important classes in this application are the *controller* and the *process*. Both inherit from class *transformer*. Between the process and the controller, a relation of type *controls* exists. Class *transformer* which contains functionality for transforming an input to an output. The process contains a sensor to interact with the real-world process it represents, whereas the controller contains an actuator to control this process.

The process is an abstraction of a real-world process. In process control theory, a number of useful process abstractions have been defined. Examples are the zero-order process, e.g. an (ideal) servo system, the first-order process, e.g. the heating of a water mass, and the second-order process, e.g. two heating tanks in sequence. Orthogonal to the order of the process, a process can have a reaction delay, i.e. a time interval between the actuation and the response of the process. These process types are archetypes and a real-world process is modelled using compositions of these process types.

Process control theory also defines three basic types of control:

- The first is *proportional* (P) control, where the difference between the actual value of the process and the desired value of the controller are linearly translated into a set value for the controller.
- The second type of control is *integrating* (I) control, where the controller not only uses the current actual value, but also the history of the process. An integrator can, for instance, level out the effects of a static disturbance in the process, whereas a P controller cannot.
- The third type is the *differentiating* (D) controller which is used to obtain a fast response to

changes in the system. It uses the derivative in time of the actual value y.

These three basic types of controllers can be composed to form more accurate (and realistic) controllers. The controller types that are used in practice are P, PI, PD and PID. In this notation, PI, for example, refers to a controller that combines proportional and integrating control.

The combination of a process and an associated controller can be used as a higher level process, which in turn can be controlled by a higher level controller. Also, processes can be combined, either parallel or sequential and the result is generally a higher order process.

4 Relations

As the term *'relation'* is used in many different contexts and any software engineer has some associations connected with the term, we have to be explicit about our use of relations. For this purpose, we define a relation as *any connection from one object (or class) O_r to another object (or class) O_s that influences the behaviour of O_r in some way.* Do note that in this definition we explicitly demand that the behaviour of the object is influenced by the relation. We are not concerned with relations that do not influence the behaviour of the object. A second aspect of this definition is the unidirectionality of a relation. The rationale for defining a relation as unidirectional is that we associate a relation with the object whose behaviour is influenced by the relation. Our experience is that only a small subset of the relation types is bidirectional in terms of our definition. These relations are modelled as two unidirectional relations.

It is our aim to present the concept of relation in a very uniform manner and to deal with each relation type in exactly the same manner. We believe that the conversion of an analysis and design model into a set of class definitions is more natural if relations can directly be expressed in the class definition.

During analysis and design, the designer can define relations of a class with other classes for three purposes. First, a relation can be used to *extend* the functionality of a class. This type of relation, we refer to as *structural relations*. Structural relations define the structure of a class and provide *reuse*. The second type of relation is defined for clients of the class. The functionality of the class is used by client objects and the class can define a *behavioural relation* with each client (or client category). Behavioural relations *restrict* the behaviour of the class. For instance, some functionality might be restricted to certain clients or in specific situations. The third type of relations that the designer can apply are *application domain relations*. Many domains have, next to reusable application domain classes, also application domain relation types that can be reused. For instance, the *controls* relation type is a very important relation in the domain of process control. The defined relation type categories are described in the following subsections

4.1 Structural Relations

Structural relation types, as described above, define the *structure* of an application. A class uses the structural relations to *extend* its behaviour and the class can be seen as the *client*, i.e. the class that obtains functionality provided by other classes. Generally, three types of structural relations are used in object-oriented systems development:

- *Inheritance*: This relation type is often seen as the central concept of the object-oriented paradigm. Inheritance allows a class to reuse the interface elements[3] and the internal state defined in the inherited class. The direction of the relation is from the inheriting class to the inherited class, as the behaviour of the inheriting class is changed.

[3]The term 'interface element' will be discussed in more detail in section 5.

relation type	*inheritance*	*delegation*	*part of*
default	Inherit	Delegate	PartOf
conditionality	ConditionalInherit	ConditionalDelegate	ConditionalPartOf
partiality	PartialInherit	PartialDelegate	PartialPartOf
conditional and partial	CondPartialInherit	CondPartialDelegate	CondPartialPartOf

Table 1: Structural relation type identifiers

- *Delegation*: A second type of structural relation is delegation, i.e. the object can delegate a message to another object while encapsulating this from the sender of the delegated message. Despite the discussion within the object-oriented research community, a number of years ago, concerning the advantages and disadvantages of inheritance and delegation, we believe that an object-oriented model should support both types of relations. The direction of the delegation relation is from the delegating object to the delegated object, as the behaviour of the delegating object is extended.

- *Part-of*: Perhaps the most fundamental relation in modelling applications is the *part-of* relation. The approach of problem decomposition and solution composition, one of the basic problem solving characteristics of humans, requires the concept of parts. This relation type is directed from the whole to the part, i.e. from an object O_{whole} to an object O_{part}. The rationale for this is that the behaviour of O_{whole} is extended by its parts and not visa versa.

The above described relation types all provide some form of *reuse*. The inherited, delegated or part object provides behaviour that is reused by the inheriting, delegating or whole object, respectively. Therefore, besides referring to these relation types as *structural*, we can also define them as *reuse* relations.

Orthogonal to the discussed relation types, which are the fundamental types of reuse relations, one can recognise two additional dimensions to structural relation types, i.e. *partiality* and *conditionality*.

- *Partiality*: The reusing object does not necessarily require the reuse of all interface elements of the reused object. Actually, it might even explicitly constrain certain interface elements to be reused. Therefore, independent of the object state or the client type, the reusing object may only want to use a part or subset of the interface of the reused object.

- *Conditionality*: When reusing an object or class, the reusing object might require that the reuse only occurs in certain states. Therefore, the *object state* is a factor in deciding whether to reuse a particular interface element. When specifying a reuse relation, the software engineer has to determine whether all interface elements of the reused object should be accessible in all object states.

We believe that a relation between two classes or objects, regardless of its complexity, should be modelled as a single entity within the model. An alternative approach would be to define a collection of orthogonal constructs and to decompose a relation between objects into instances of each orthogonal construct as is done in, e.g. [3]. This approach, however, does not represent a conceptual entity in analysis and design as an entity in the language model.

The different aspects of a structural relation, i.e. its type, partiality and conditionality, form a three-dimensional space which contains all possible combinations. In compliance with our modelling principle, we define a relation type for each combination. This results in twelve structural relation types. These structural relation types are shown in table 1.

4.2 Behavioural Relations

In the previous section, relation types were discussed where the object containing the relation is the *client*, i.e. the object 'extends' itself with the structural relations. The structural relations construct an object by extending it with the behaviour of other classes. In this section, relation types between the object (the server) and its clients are discussed. These relation types, generally, constrain the access of clients to the behaviour of the server object in some way.

In order to keep the type and number of clients of the object open ended, we define *client categories* and for each message is determined whether the sender of the message is a member of a client category. The message is then subject to the behavioural relation(s) defined for that client category.

A relation between the object and a client category can define several types of behavioural constraints. A behavioural relation, in our model, incorporates four types of constraints, i.e. based on the *client*, the *state*, *concurrent access* and *time constraints*. Below, these types of constraints are discussed in more detail:

- *Client-based access*: The simplest form of behavioural constraint relation is the restriction of access to interface elements based on the client category. That is, some clients are allowed to access certain interface elements, while this is forbidden for others.
- *State-based access*: This is used to restrict the access to interface elements by certain clients when the object is in a specified state: This is generally used to favour other clients. For instance, if a stock object has a small number of elements left, access might be restricted for occasional customers, thereby favouring the more important, frequent customers. Do note that state-based access is different from conditionality in the structural relation types. In structural relations, conditionality is used to restrict access to *reused* interface elements when the object is in a certain state. That is, in certain object states, interface elements are not part of the *structure* of the object. State-based access for clients is used to restrict access for clients, but the interface elements remain part of the structure of the object.
- *Concurrency*[4]: An object, operating in a concurrent environment, can be addressed simultaneously by several clients. To maintain an internally consistent state the object needs to synchronise the concurrent messages.
- *Real-time*: The object can associate real-time constraints with messages. The message obtains a time interval within which the request needs to be executed. The reason for including real-time constraints is that although the first-come-first-served ordering might provide optimal throughput, it is generally not the optimal strategy for an application which interacts with a user or other systems. A mechanism to provide relative ordering of concurrent execution can be very beneficial.

In this paper we propose an alternative approach to the definition of modelling constructs when compared the conventional approaches. Rather than defining 'clean', orthogonal constructs, we try to define constructs that correspond to conceptual entities. Hence, we believe, supported by the object-oriented analysis and design methods, that the concept of a relation between objects is a conceptual entity and the different aspects of this relation should be defined as part of this relation, rather than as several unrelated orthogonal constructs that, when combined, provide equivalent behaviour.

Analogous to the structural relations, the behavioural constraints form orthogonal dimensions in a four dimensional space. However, as we base ourselves on the notion of relations between objects, we

[4]Do note that we use the term 'concurrency' in its literal meaning, i.e. concurrent execution of two methods within an object. This differs from the use of the term in e.g. [3], where the term 'concurrency' includes what we refer to as 'state-based access'.

No factor	*One factor*	*Two factors*	*Three factors*
RestrictClient	RestrictState	RestrictStateAndConc	RestrictStateConcAndTime
	RestrictConc	RestrictStateAndTime	
	RestrictTime	RestrictConcAndTime	

Table 2: Behavioural relation type identifiers

require that a client category is always specified. The resulting three dimensions can be part of the relation, but this is not required. For each location in this space, a relation type is defined. In table 2 the different relations types are shown.

4.3 Application-Domain Relations

Next to the general relation types, described in the previous two sections, one can also identify application domain specific relation types. From our experiences in modelling object-oriented systems, we have learned that many domains, next to characteristic classes, also have characteristic relation types that ought to be captured in framework and application development.

Within the context of this paper, it is impossible to define the relevant relation types for a large number of application domains. The development of frameworks has proven to be a difficult and tedious process and we have no reason to assume that the development of a framework containing relation types will be any easier. Instead, we will illustrate our approach by an example from an example domain, i.e. *process control*, described in section 3.

When modelling a process control application framework, the most important objects are the controller and the process. In figure 2, an analysis model of a part of a typical process control application is shown. When modelling relations between objects as first-class entities, the '*controls*' relation between the controller and the process is a typical example of an application domain specific relation type. A *'controls'* relation can be of several types which all behave differently. For the discussion in this paper, we identify four control relation types:

- **P**: Proportional control uses an amplification factor K_c to amplify the difference between the actual value y and the set value r, which is than used to react against the difference. Proportional control is characterised as $H_c = \frac{1}{1+K_c}$, where H_c is the transform ratio. We define a relation type P as a generic proportional control relation type with a configuration parameter K_c to indicate the amplification factor. Other configuration parameters are r, the set variable, which is part of the controller object, the process input variable x, which sets the input of the controlled process, and the process output variable y, which is the actual output value of the process.

- **PI**: Integrating proportional control contains, next to a proportional part, an integrating part. This type of controller is characterised as $H_c = K_c(1 + \frac{1}{j\omega\tau_i})$, where H_c is the transform ratio of the controller, K_c is the aforementioned amplification factor, and τ_i is configuration parameter for the integrator. The PI controller relation type, therefore, uses the parameters K_c, τ_i, r, x and y.

- **PD**: Differentiating proportional control is used in cases where a quick response is required. The PD controller is characterised as $H_c = K_c(j\omega\tau_d + 1)$, where τ_d is the configurating parameter for the differentiator. The PD controller relation type uses the parameters K_c, τ_d, r, x and y.

- **PID**: In certain processes, both the advantages of the integrator, i.e. zero statistical deviation, and of the differentiator, i.e. quick response, are required to obtain satisfactory process control. This PID controller is characterised by $H_c = K_c(1 + \frac{1}{j\omega\tau_i})(j\omega\tau_d + 1)$. The configuration parameters for the PID controller are K_c, τ_i, τ_d, r, x and y.

The precise semantics of the specificed relation types can be found in most process control books, e.g. [9]. In section 5.5 some examples of the use of *controls* relations are given.

4.4 Concluding Remarks

In this section we have introduced a categorisation of relations between objects into *structural relations*, *behavioural relations* and *application-domain relations*. Rather than defining a 'clean' set of orthogonal constructs that can be composed, we propose to define a relation type for each conceptual relation type that is identified by the software engineer.

One can recognise the similarity between our approach and the notion of defining frameworks. When defining a framework, the software engineer conceptualises important entities in the domain as classes. We propose a similar approach, but now for the relation types within a domain.

However, different from other approaches, e.g. [15], we do not model relations as independent objects within the application, but extend the conventional object model with a new component that allows the designer to make the relation an integrated part of the object whose behaviour is extended or influenced. The precise semantics of this new object model component are explained in the next section.

5 Layered Object Model

The layered object model (LAYOM) is an extended object model defined at the University of Karlskrona/Ronneby as a vehicle for research in extended expressiveness of object-oriented language models. The object model of LAYOM incorporates components that are not found in the conventional object model. One of these components, the *layer* concept, supports the first-class representation of relations. In this section, first LAYOM is introduced. Then the layer concept is defined. Lastly, the structural, behavioural and application domain relation types are defined.

5.1 Basic Object Model

The layered object model (LAYOM) is an extended object model, i.e. it takes the conventional object model as a basis and extends it to improve the expressiveness. As is shown in figure 3, an object in LAYOM consists of five major components, i.e. *variables* (nested objects), *methods*, *states*, *conditions* and *layers*. A LAYOM object has an *interface*, i.e. a set of interface elements, consisting of the methods, the states and the conditions of the object. These three components share a common name space. Each interface element can be invoked by a client object, provided that access to the interface element is not prohibited. Below, the functionality of each object model component will be described.

A *state*, as defined in LAYOM, is a dimension of the *abstract object state*. The abstract object state of an object is an externally visible abstraction of the internal, concrete state of the object. Each dimension has a domain and the state expression associated with the state 'maps' the concrete state (or a part of it) to the domain associated with the state. Within the LAYOM environment, a domain is basically a subclass of class *Domain*. This allows the designer to define his own domains and to use them freely. The rationale for defining an abstract object state and its precise semantics is described in [6], but it can be summarised as a systematic and structured approach to make the *conceptual* state of the object accessible at the interface, as an alternative for the ad-hoc fashion in which object state is made accessible in conventional approaches.

As a part of the abstract object state, the designer can define so-called *active states*. An active state is defined based on the static abstract state defined for the object and is a derivation function of (a part of) the static abstract state in time. For example, when the output value of an process is part

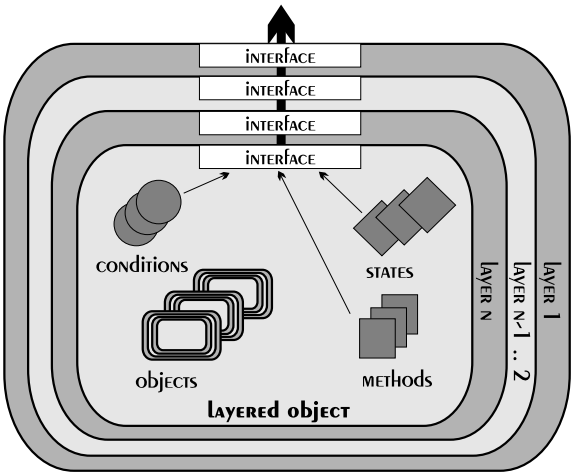

Figure 3: Components of a LAYOM object

of the static abstract state, a possible active state could be the derivative output value in time which could be used by a differentiating controller type. Based on the active state principle, the designer can define higher-order state spaces that allow first-class representations of otherwise continuously changing aspects of the system.

Client categories are defined in LAYOM using *conditions*. A condition, just as a state, can be seen as a specialisation of a general method. A condition has a single, implicit argument *msg*, i.e. the message for which is determined whether its sender belongs to the client category defined by the condition. The return value of the condition is restricted to class *Boolean*.

The variables of the object are other objects nested within the object in which they are declared. Each nested object has an abstract object state which is visible on its interface. The abstract object states of the nested objects form the concrete state space for the encapsulating object. The state definitions in the class description define an abstraction, i.e. the abstract object state, of the concrete object state of the encapsulating object. Variables are declared, as in many other languages, by defining the name and the class of the nested object.

The methods of the object are defined as in most object-oriented languages. The scope of the method is inside-out, and consists of the local variables declared by itself, the arguments of the method, the variables defined by the object the method is part of. Also the objects defined at the same level as the object containing the method and all objects at the subsequent levels are visible. Basically, each encapsulation boundary can be 'passed' from the inside, but is 'restricted' from the outside.

The *layers* encapsulate the object so that messages sent to the object or sent by the object itself have to pass all layers. Each layer can change, delay, redirect or respond to a message or just let it pass. We use layers for the representation and implementation of relations which the encapsulated object has with other objects and client categories. In the next section, layers are discussed in more detail.

As an example, the class definition of class *Transformer* (see figure 2) is shown in figure 4. Class *Transformer* has two instance variables and two methods. In addition, it defines a state *outputOK* which maps a part of the concrete state of the object, i.e. the value of the outputValue instance variable, to a user defined domain *RangeEnumeration* which is defined as (*too-low*, *low*, *normal*, *high*, *too-high*). Also, the class defines a condition (client category) *operator* which determines whether the sender of a message is an instance of class *Operator* or one of its subclasses. The layer *lay_bc* is an instance of the *RestrictClient* behavioural relation type which restricts access to the method

```
class Transformer
  layers
    lay_bc : ByClient(operator, setInputValue);
  variables
    inputValue : Real;
    outputValue : Real;
  methods
    setInputValue (newValue : Real) returns Boolean
    begin
      ...method code ...
    end;
    transformValue returns Real
    begin
      ...method code ...
    end;
  states
    outputOK returns RangeEnumeration
      // RangeEnumeration is a user defined domain class
    begin
      ...state expression ...
    end;
  conditions
    operator
    begin
      sender.subClassOf(Operator);
    end;
end Transformer;
```

Figure 4: Class *Transformer*

setInputValue to members of the client category *operator*.

5.2 The Layer Concept

As is shown in figure 3, a LAYOM object is generally encapsulated by *layers*. Messages sent to or sent by the object need to pass the layers. A layer can intercept a message and react in a predefined manner to the receipt of the message. This allows for the representation of *relations* by layers. The relation types that were identified in section 4 are implemented as layer types in the LAYOM object model.

A message that is sent to the object has to pass the layers that encapsulate the object, starting at the outermost layer and working his way in. Each layer receives the message and inspects it in order to determine what it will do with the message. A layer can delay the message until some precondition is fulfilled or it can change the message contents. Other possibilities are that the layer itself replies to the message or that it rejects the message, generating an exception. Basically, each layer object has the opportunity to handle a message in virtually any way it likes. However, in practice, a layer will pass most messages on to the next layer.

Analogous to a message sent *to* the object, a message sent *by* the object has to pass all the layers starting at the innermost layer. Again, each layer has the possibility to treat outgoing messages in any way it feels appropriate. Do note that each layer, either in response to a message it is requested to evaluate or otherwise, can send messages to the object it encapsulates. For instance, a layer can request the value of a state or a condition.

If a message arrives at a layer, the message is reified by the layer. This results in a passive object of class *Message* which can be investigated by the layer to determine its action. The message object

```
class Process
  layers
    ...
    /* structural relations */
    inh : Inherit(Transformer);
    aS : PartOf(Sensor);
  ...
end Process;
```

Figure 5: Structural relation layers of class *Process*

has fields containing a reference to the sender of the message, the selector, a reference to the receiver and the arguments to the addressed method. The layer will, depending on its type, the state of the object it encapsulates and the message, behave in a predefined manner.

To illustrate this we describe the behaviour of a layer of type *Inheritance*. Class *Process* has a layer `inh : Inherit(Transformer)`, meaning that it has an inheritance relation with class *Transformer*. The interface of class *Transformer* consists of $I_{Transformer} = (i_1, \ldots, i_n)$. The layer *inh* represents this relation. Upon instantiation of an instance of class *Process*, layer *inh* is also instantiated and the layer will create an instance of class *Transformer* as a part of itself. When a message is sent to the instance of class *Process*, it will have to pass the layer *inh*. The layer will reify the message and check if the selector matches with one of the interface elements, i.e. $selector \in I_{Transformer}$. If the message matches, it is forwarded to the instance of class *Transformer* which is part of the layer, otherwise it is passed on to the encapsulated object.

In this article we focus on the use of layers for the representation of relations between objects. However, layers can also be used to extend the interface of the object. For instance, a layer of type *Condition* can be used to define a new client category which was not defined within the object itself. From the perspective of a layer, it makes no difference whether it is in-between other layers, the most outer layer or the most inner layer. An object is defined so that at the outside of each layer, one views an object with an interface consisting of methods, states and conditions. For example, if an additional layer of type *RestrictClient* is defined around the object which makes use of condition layers, the *RestrictClient* layer notices no difference between conditions defined by encapsulated layers or conditions defined within the kernel object.

Each layer type has an identifier, its own specification syntax and semantics. When defining a new layer type, the software engineer specifies these aspects. In the prototype implementation, the software engineer is able to reuse existing layer specifications using the parser delegation concept [5].

5.3 Structural Relations

The structural relations within LAYOM are not implemented within the object model. The conventional object model implements inheritance and part-of relations as part of the object model, but has no support for other relations types. As we consider this to be problematic, LAYOM provides a different approach to represent relations, i.e. *layers*. Based on the analysis of structural relation types in section 4.1, LAYOM supports the relation types defined in table 1.

To illustrate this, we use, as an example, class *Process* (see figure 2) which has a number of relations with other classes. In figure 5, the layer specifications for the structural relations of class *Process* are shown. The first layer *inh* represents an *inheritance* relation with the superclass of class *Process*, i.e. class *Transformer*. The second layer, i.e. *aS*, defines an instance of class *Sensor* as part of class *Process*.

In table 1, 12 structural relation types are defined. We could specify all types, but for reasons of

space, we will limit ourselves to a subset. The specification of the other types is implicated by the specification of the subset.

- **Inherit**: The syntax of the relation is very simple:
  `<identifier> : Inherit(<class-name>)`
  The layer will intercept all messages to the object that are implemented by class `<class-name>` and redirect these messages to an instance of that class, created by the layer.

- **Delegate**: The syntax of this layer is almost the same as that of *Inherit*:
  `<identifier> : Delegate(<object-name>)`
  The layer will intercept messages sent to the object and redirect those messages that are implemented by the object `<object-name>` to that object. This object must be in the scope of the object containing the layer.

- **PartOf**: The syntax is:
  `<identifier> : PartOf(<class-name>)`
  The layer will instantiate an object of class `<class-name>`. The layer will intercept messages sent by the object it encapsulates that are addressed to object `<identifier>`, i.e. the name of the layer is used as the name of the part object is represents.

- **PartialInherit**: This relation type is used in the situation where the inheriting object only wants to inherit a subset of the interface of the inherited class. The syntax is:
  `<identifier> : PartialInherit(<class-name>, included-set, excluded-set)`
  Both `included-set` and `excluded-set` can be a list of one or more methods names or a star, i.e. $\star$. The star is used as a wild-card and represents all interface elements. However, the star is preceded by an explicit list. Otherwise, if the included set and the excluded set overlap, the intersection of the interface elements will be treated as only part of the included set.

- **ConditionalDelegate**: Sometimes it depends on the state of the object whether the relation should be active or not. This relation type defines conditionality on the delegation relation, i.e. the delegation will only occur if the object is in the specified state. The syntax is defined as follows:
  `<identifier> : ConditionalDelegate(<object-name>, active-boolean-expression, passive-boolean-expression)`
  If the active expression evaluates to true, the message will be delegated. If the active expression only contains the keyword `true` and the passive expression evaluates to true, the message will not be delegated to the object.

5.4 Behavioural Relations

The behavioural relation types are used to define the behaviour of the object for a particular client category. In table 2, the identified behavioural relation types are presented. To illustrate the use of this type of relations, we again use class *Process*. Some requirements of class *Process* not mentioned in section 3 are that 1) the *setInputValue* method should only be accessed by instances of class *Controller*, 2) the *Process* can only be accessed when it is in the *Operational* state, 3) only one client will be serviced by an instance of class *Process* and 4) the *setInputValue* method has a deadline of 10 time units.

When analysing these requirements, one can derive two relations. The first relation is with the *controller* client category. This relation incorporates client-based access and a time constraint. Therefore, its relation type is *RestrictTime* and the client category is *controller*. The second relation type is more general and incorporates state-based access and a concurrency constraint. The type of the relation is *RestrictStateAndConc* and the client category is *Any*. *Any* is used to address all clients. Taking a set-based view to the clients of an object, the *Any* client category is the set encapsulating

```
class Process
  layers
    /* behavioural relations */
    bt : RestrictTime(controller, setInputValue finish-latest 10);
    bsc : RestrictStateAndConc(Any,
            * when (Operational=True) in-which-case delay 1 active-threads);
    /* structural relations */
    inh : Inherit(Transformer);
    aS : PartOf(Sensor);
  ...
end Process;
```

Figure 6: Behavioural relation layers of class Process

all clients categories. Any explicitly defined client category C is a subset of this set, i.e. $C \in Any$. In figure 6 the behavioural relations of class *Process* are shown.

In table 2, 8 behavioural relation types are defined. Again, for reasons of space, the description is limited to a subset. The specification of the other types is implicated by the specification of the subset.

- **RestrictClient**: This relation type is used to restrict access to parts of the interface of the object for certain client types. The syntax is:
  `<identifier> : RestrictClient(<condition>, access-set, restrict-set)`
  The interface elements in the `access-set` are accessible to the client category indicated by `condition`. The elements in the `restrict-set` obviously not. As described before, the `access-set` has a preference over the `restrict-set`, unless the star is used in the `access-set`.

- **RestrictState**: When the access to the interface by a client category is also dependent on the state of the object, one should use this relation type. The syntax is:
  `<identifier> : RestrictState(<condition>, (`*accept* `access-set`
  `[` *when* `|` *unless*`] boolean-expression [`*otherwise* `|` *in-which-case*`] [`*reject* `|` *delay*`])+)`
  Here the '+' indicates that the part between the brackets is present one or more times.

- **RestrictConc**: If not restricted, the execution of methods within the object is fully concurrent. If this could lead to inconsistency of the object, the concurrency needs to be restricted. The *RestrictConc* relation type is defined for this purpose. The syntax is:
  `<identifier> : RestrictConc(<condition>, ([<number>` *active-threads* `|`
  `exclusion-set` *exclusive*`])+)`
  For each client category, the object can restrict the concurrency to either a `number` of concurrent threads, or it can define one or more exclusion-sets. Each `exclusion-set` contains two or more interface element identifiers that are not to be executed concurrently.

- **RestrictTime**: The default deadline on a message is ∞. If a message from a client category has a certain deadline, this has to be specified in the relation to that client category. The syntax is:
  `<identifier> : RestrictTime(<condition>, (access-set` *start-earliest* `<time>`
  finish-latest `<finish>)+)`
  The semantics are that for the client category specified by `condition`, the software engineer can specify one or more interface sets. Each `interface-set` consists of a list of interface elements or a star, an earliest start time `start` and a latest finish time `finish`.

- **RestrictStateConcAndTime**: This is the most complex relation type as it composes the behaviour of the four types of behavioural constraint. The syntax is:

<identifier> : RestrictStateConcAndTime(<condition>, ([access-set | ⋆ *except* restrict-set] [<number> *active-threads* | *exclusive*] [*when* | *unless*] boolean-expression [*otherwise* | *in-which-case*] [*reject* | *delay*] *earliest-start* <time> *latest-finish* <time>)+)

For the client category denoted by **<condition>**, one can specify the state in which the access-set becomes accessible, the concurrency within that set and the real-time constraints of this relation.

5.5 Application Domain Relations

The application domain relation types are, obviously, not predefined. The software engineer can define new relation types and apply these relation types in class definitions as any relation type. When defining a new relation type, it is basically the name, the syntax of the initialisation and the semantics that have to be defined. These aspects are described in more detail in section 5.6.

To illustrate the use of application domain relations, we continue with the *controls* relation types, described in section 4.3. In figure 2, class *Controller* has a *controls* relation with class *Process*. Depending on the type of *controls* relation between the controller and the process, i.e. *P*, *PD*, *PI* or *PID*, a control relation type is defined between the two classes. For example, in case of proportional control, a *P-control* relation type is selected. Each relation type has its initialisation syntax which has to be defined as part of the class definition containing the relation. Below, the identified relation types are discussed in more detail:

- **P-control**: The initialisation syntax is:
  <identifier> : P-control(<set method>, <process.input-method>, <process.output-method>, <multiplication factor>)
  The set method is a method of the *Controller* class that is used to set the required value of the process. The process.input-method is the method at the process object that sets the required value of the process, whereas the process.output-method is the method of the process that gives the actual process value. The multiplication factor is a proportional control constant that indicates how strong a difference between the required and the actual value of the process should be applied in the set value of the process.

- **PI-control**: The initialisation syntax is:
  <identifier> : PI-control(<set method>, <process.input-method>, <process.output-method>, <multiplication factor>, <integration factor>)
  All but the last elements are the same as the ones defined for *P-control*. The integration factor is a constant for the integrating control. It basically indicates how strongly past deviations from the set value are incorporated in the control signal to the process.

- **PD-control**: The initialisation syntax is:
  <identifier> : PD-control(<set method>, <process.input-method>, <process.output-method>, <multiplication factor>, <differentiation factor>)
  Again, all but the last elements are the same as the ones defined for *P-control*. The differentiation factor is a constant for the differentiating control. It basically indicates how strongly the angle of the current deviation from the set value is incorporated in the control signal to the process.

- **PID-control**: The initialisation syntax is:
  <identifier> : PID-control(<set method>, <process.input-method>, <process.output-method>, <multiplication factor>, <integration factor>, <differentiation factor>)
  All but the last two elements are the same as the ones defined for *P-control*. The last two elements are defined as for *PI-control* and *PD-control* respectively.

```
class Controller
  layers
    c1 : P-control( setInputValue, pp.setInputValue, pp.outputValue, 100);
    c1 : PI-control( setInputValue, pp.setInputValue, pp.outputValue, 100, 5);
    c1 : PD-control( setInputValue, pp.setInputValue, pp.outputValue, 100, 0.74);
    c1 : PID-control( setInputValue, pp.setInputValue, pp.outputValue, 100, 5, 0.74);
    /* Structural relations */
    inh : Inherit(Transformer);
    anA : PartOf(Actuator);
  ...
end Controller;
```

Figure 7: Control relations of class Controller

In figure 7, the possible control relations are shown. The method `setInputValue`, in these examples, is a method of the *Controller* class that sets the required value for the process. The object `pp` is an external object of class *Process*. Its methods `setInputValue` and `outputValue` are used as the process input variable and process output variable, respectively. The constants `100`, `5` and `0.74` are the multiplication, integrating and differentiating constant, respectively.

These relation types contain all the complex aspects of the respective types of control. If, as in the conventional object model, the control behaviour would be implemented as part of class *Controller*, the control specific behaviour would be mixed with the other functionality of class *Controller*. Also, the functionality of the relation would have become part of the object and the explicit modelling relation between the analysis and design model and the implementation model would have been lost. This, again, indicates the importance of representing a relation as an entity in the object model.

5.6 Implementation

The layered object model, as described in this paper, is supported by a prototype development environment, called ATOM developed at the University of Karlskrona/Ronneby. ATOM – which stands for 'A Translator for the layered Object Model' – consists of four major parts:

- **Layered Object Model Compiler (LOC)**: The LOC translates a LAYOM class into a corresponding C++ class. Also, in LAYOM the software engineer can define an 'application' which is a specification of a number of objects and some code. The LOC translates an LAYOM application into a corresponding C++ main and invokes the C++ compiler and linker to generate an executable.
- **ATOM Execution Environment (ATEE)**: ATOM is constructed and operating in a Sun Sparc and Solaris 2.3[5] environment. The ATEE provides the run-time environment for LAYOM applications. This environment supports transparent distribution and concurrency. The distribution is achieved through the use of UNIX[6] sockets and concurrency is achieved through the use of threads which are part of the Solaris operating system. The ATEE allows the user to start multiple applications, i.e. C++ executables, at multiple machines. In each application, the objects can call objects that are not defined within the application. In that case the ATEE will search for the objects in the other applications. Real-time constraints and dynamic load balancing are planned to be added in the future.
- **Layer Compiler Compiler (LCC)**: In this article we have stressed the importance of the ability to define new abstractions. For instance, in section 5.5 four new relation types are added

[5]Sun and Solaris are trademarks of Sun Microsystems inc.
[6]Unix is a trademark of AT&T Bell Labs.

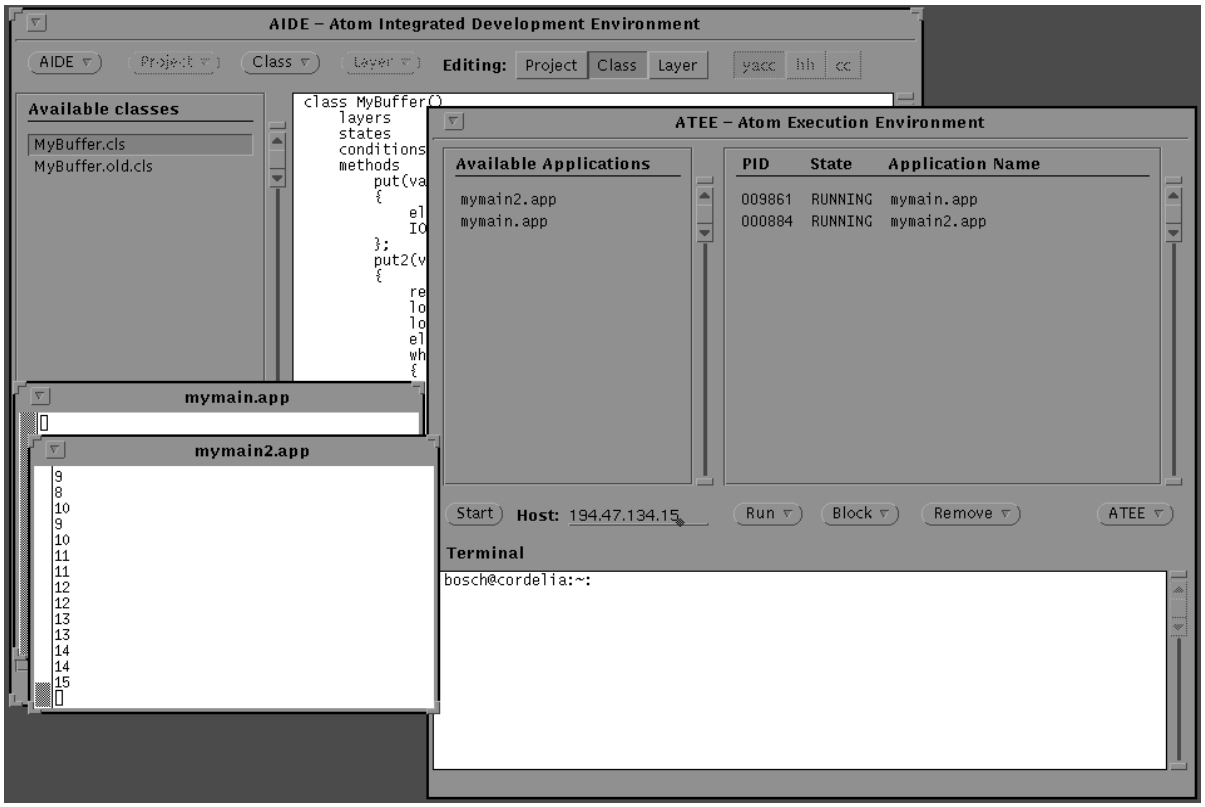

Figure 8: ATOM AIDE and ATEE

to the system. In LAYOM these relation types are implemented as *layers* encapsulating the object. For each new relation (or layer) type, the software engineer has to define an identifier for the type, the syntax of the initialisation block and the semantics of the layer type. This functionality is offered by the LCC which allows one to specify the syntax of a new layer using an extended YACC syntax and the semantics in the semantic part of the production rules.

Each layer can define its independent syntax. This would, in a naive implementation, require the class parser to be extended every time a new layer type is defined, as the software engineer can use a new layer type as soon as it is defined. To address this problem, we have introduced a novel mechanism, i.e. *parser delegation*, which allows for multiple parsers to be active during parsing. The details of *parser delegation* are described in [5].

- **ATOM Integrated Development Environment (AIDE)**: To facilitate the software engineer, we have constructed an integrated development environment AIDE which allows the software engineer to define LAYOM classes, applications and new layer types in an integrated manner and to experiment with defined applications using the ATEE. The AIDE basically is a shell that provides convenient access to the LOC, ATEE and LCC and that handles projects, class definitions and other data in a user-friendly manner.

In figure 8, a part of the ATOM environment is shown. In the background the AIDE is visible and the ATEE is in the front. The small windows are input/output windows of two applications that run distributed and communicate with each other.

6 Related Work

Expressing relations within the object-oriented model has been proposed previously by Rumbaugh [15]. However, his intention was to model relations as separate set entities that are not related to any object. Objects that are involved in a relation can be registered at the relation object. His approach aims primarily at facilitating the use of relational database systems in object-oriented applications.

Compared with Rumbaugh's approach, our approach to modelling is different in both the intention and the approach. Our aim is to simplify the transformation from an analysis and design model into the object model by providing means to represent relations directly in the object model. Second, our approach is to model relations as layers encapsulating the object whose behaviour is influenced by the relation, rather than modelling relations as separate entities next to objects. Third, we do not aim at object-oriented modelling techniques for relational database applications, but at extending the set of relation types that can directly be represented in the object model.

The layered object model can be viewed as an object model with reflective capabilities, in that it engages message reflection to achieve its extended functionality. However, it is different from reflective object models as described in e.g. [13] that provide full reflection. LAYOM provides only message reflection and clearly separates the specification of meta-level objects, i.e. layer (or relation) types from the 'normal' implementation.

Another example of a limited reflective object model is the composition-filters (CF) object model [3], which partially inspired our work on the layered object model. The CF model applies input- and output-filters which intercept incoming and outgoing messages, respectively. The CF model defines filter types like *error*, *wait* [3], *dispatch*, *meta* [2] and *real-time* [1] and each filter type provides one specific (orthogonal) type of behaviour. When comparing LAYOM and the CF model, one can observe at least three major differences. First, filters do not support the representation of a relation as a single entity, but, depending on its requirements, a relation will be divided over multiple filter specifications. Second, filters do not support the specification of structural relations at the filter level although they do provide means for expressing conditionality and partiality. Thirdly, to the best of our knowledge, the CF model does not support the specification of application-domain abstractions.

The layers of a LAYOM object may seem similar to encapsulators [14]. However, encapsulators are implemented as objects in the application that have a reference to the encapsulated object. Encapsulators rely on the Smalltalk-80 *doesNotUnderstand:* mechanism and encapsulators are mainly used to extend methods with pre- and post-actions, whereas layers are used to represent relations between objects. In addition, a layer is a component of the object model and has an configuration syntax which configures the layer, whereas encapsulators are regular objects.

Contracts [11, 12] offer a mechanism to model interdependencies between objects. Contracts are different from the LAYOM approach in that the interdependencies are represented as separate entities in the applications, rather than being associated with the object whose behaviour is influenced.

7 Conclusion

Relations between objects are very important modelling entities that, unfortunately, are not supported by the conventional object model that is generally used to implement the analysis and design model. This results in several problems as there are *the lack of uniformity*, *the loss of a source for reuse* and *diminished traceability*. In this paper we have analysed relations between objects and defined three categories of relations: *structural*, *behavioural* and *application-domain* relation types. To provide a solution to the identified problems, we proposed the *layered object model* (LAYOM) which supports the representation of relations through the layer concept as part of the object model. We have illustrated the translation of an analysis and design model into LAYOM classes using the *process control domain* as

an example. Each relation in the analysis and design model can directly be represented in the object model.

When evaluating our approach with respect to the identified problems, we can conclude that LAYOM does not suffer from these. It provides a highly uniform model for expressing relations between objects and, because of the possibility to define application-domain relation types, is it possible to reuse application-domain relation types. As relations are represented as identifiable entities, traceability of relations is not inhibited.

The layered object model is supported by a prototype environment consisting of LOC, a translator translating LAYOM classes and applications into C++ classes and main functions, ATEE, an execution environment providing concurrency and transparent distribution, LCC, a layer compiler that allows for the definition of user defined layers, like application-domain relation types, and AIDE, an integrated development environment that integrates the functionality of the aforementioned tools.

Future work will include the continued evaluation of application-domain relation types and the search for other abstractions that might improve the expressiveness of the object model. With respect to the ATOM prototype, we aim at extending the functionality with automated *dynamic load-balancing* and support for *real-time constraints*.

Acknowledgements

I would like to thank Lodewijk Bergmans, Lennart Ohlsson and Fredrik Ygge for their comments and the members of the student project group ATOM for implementing the translator and environment for LAYOM.

References

[1] M. Aksit, J. Bosch, W. vd Sterren, L. Bergmans, "Real-Time Specification Inheritance Anomalies and Real-Time Filters," *Proceedings ECOOP'94*, pp. 386-407, LNCS 821, Springer-Verlag, July 1994.

[2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, "Abstracting Object Interactions Using Composition-Filters," in *Object-based Distributed Programming,* R. Guerraoui, O. Nierstratz, M. Riveill (eds.), LNCS 791, Springer-Verlag, 1994.

[3] L. Bergmans, "Composing Concurrent Objects," *PhD Dissertation*, Department of Computer Science, University of Twente, The Netherlands, 1994.

[4] G. Booch, *Object Oriented Design (with applications),* Benjamin/Cummings Publishing Company, Inc., 1990.

[5] J. Bosch, "Parser Delegation – An Object-Oriented Approach to Parsing," to be published in *TOOLS Europe '95.* Also Research Report 7/94, University of Karlskrona/Ronneby, September 1994.

[6] J. Bosch, "Abstracting Object State," submitted to *Object-Oriented Systems,* December 1994. Also Research Report 10/94, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, December 1994.

[7] J. Bosch, "Paradigm, Language Model and Method," to be presented at the *ICSE-17 Workshop on research issues in the intersection of Software Engineering and Programming Languages,* April 1995.

[8] D. de Champeaux, D. Lea, P. Faure, *Object-Oriented System Development,* Addison-Wesley, 1993.

[9] J.C. Cool, F.J. Schijff, T.J. Viersma, *Regeltechniek,* Agon Elsevier, 1969.

[10] D. Embley, B. Kurtz, S. Woodfield, *Object-Oriented System Analysis - A Model-Driven Approach,* Prentice Hall, 1992.

[11] A.R. Helm, I.M. Holland, D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," in *Proceedings OOPSLA '90,* pp. 169-180, Ottawa, 1990.

[12] I.M. Holland, "Specifying reusable components using Contracts," in *Proceedings of the 1992 European Conference on Object-Oriented Programming,* pp. 287-308, Utrecht, The Netherlands, June/July 1992.

[13] G. Kickzales, J. des Rivières, D.G. Bobrow, *The Art of the Metaobject Protocol,* The MIT Press, 1991.

[14] G.A. Pascoe, "Encapsulators: A New Software Paradigm in Smalltalk-80," in *Proceedings OOPSLA'86*, pp. 341-346, Portland, 1986.

[15] J. Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language," *Proceedings OOPSLA '87*, pp. 466-481, ACM SIGPLAN Notices, Volume 22, Number 12, December 1987.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design,* Prentice Hall, 1991.

[17] R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software,* Prentice Hall, 1990.