System Design by Composing Structures of Interacting Objects

Egil P.Andersen[1] & Trygve Reenskaug[2]
Department of Informatics, University of Oslo
P.O.Box 1080, Blindern, 0316 Oslo, Norway

Abstract

This paper describes the outline of an object-oriented design technique denoted role modeling, emphasizing the ability to compose parts of a design. The purpose of role modeling is to achieve separation of concerns, allowing the designer to consider different aspects, or the same aspect at different levels of detail, more or less independent of other aspects of the overall design.

A role model represents the concept of a structure of communicating objects; each object being represented by a role to be 'played' in the context of this role model. Each role model is considered a design of a separate aspect of some overall design. Composition of designs is achieved by synthesizing roles in several role models, constructing more aggregated and specialized roles and role models.

Keywords: O-O Design, Interaction-Oriented Design, Role Modeling, Object Composition

1 Introduction

Our basis is best presented by the words of Beck & Cunningham[Helm], '*...no object is an island*'. We consider real-world concepts to consist of several mutually cooperating and interacting entities, not of stand-alone entities existing in a vacuum independent of other entities in the same domain of interest.

Assuming this view to be appropriate for a particular design task at hand, the issues we will address are the following:

- How may such concepts be represented; i.e how do we describe which objects interact in a particular task or activity, and which messages do they send to and receive from each other?
- If having a set of basic designs, how do we create either more aggregated or more detailed designs? Furthermore, how do we hide irrelevant details in our designs and how may we view them at different levels of detail; i.e how do we create abstractions?
- How do we recognize the requirements of classes supposed to implement our design?
- How do we make objects execute the behaviour described in a design, and how do we know which objects may participate in a particular design?

The design approach presented here is related to the ideas of considering objects as 'playing' different roles in different contexts [Arapis][Pernici], and of specifying behavioural

[1]Email: egil@ifi.uio.no
[2]Email: reenskaug@taskon.telemax.no

compositions in object-oriented systems by *contracts* [Helm]. These ideas are also denoted *interaction-oriented* or *responsibility-driven* design [Wirfs-Brock89], or *'points of view'* on objects [Shilling] among others. We believe the novelty of our approach to be the use of synthesis, as described below.

2 Role Modeling

2.1 The Purpose of Role Modeling

Our response to the first issue above is the role modeling technique as presented in [Reenskaug2]. Before addressing the other issues we will give a brief summary of that presentation.

A *role model* is the unit of design. It comprises two or more interacting entities denoted *roles*. Each role is considered a requirement/responsibility of objects participating in the actual execution of the behaviour described in the role model. An object is said to be able to *'play'* a particular role if it satisfies the requirements of this role. Furthermore, a role may be considered a documentation of a structural and behavioural view/aspect/part of the object playing this role in context of the particular role model. In the following we will say that an object *configures* a particular role if it 'plays' that role in a particular role model.

Where are the benefits of the role modeling approach? In most object-oriented programming and design systems, a class is viewed as an abstract data type offering an interface of methods/messages that can be applied to objects of this class. Usually the methods in the interface are accompanied by a formal or informal description of its behavioural effect. Methods corresponding to an activity, function or task involving only the object itself and its attributes are usually easily comprehended by such a description. This is often not the case if a method is part of an activity involving several communicating objects. Then the description of the behavioural effect is distributed over several classes of the participating objects. To comprehend such an activity the user has to consider several classes at the same time. This is complicated by objects of each class participating in several possibly independent activities, and the description of these activities being intermingled in each class.

The objective of role modeling is to aid comprehension and documentation of such activities by supporting separation of concern. The designer can either construct a role model for each activity or task carried out in the overall system, or construct several role models for the same activity at different levels of detail. By considering a single role model, the designer is able to consider one activity and its participating roles at the time, without being burdened by the entire model and all of its rather irrelevant details. Certainly, every aspect of the model as a whole is connected and related, but usually there are several aspects that to a large extent may be comprehended separately. Only when composing separate role models is it necessary to take their mutual dependencies into consideration.

We should emphasize that in general, the mission of role modeling is to reduce complexity when doing 'large-scale' design; i.e complexity due to the *size* of the design task. This is done by supporting separation of concern, a flexible design approach and reusable designs. At present the role modeling technique has no contribution if the complexity of the task is due to the complexity of the message interactions themselves. Then specification techniques

and tools supporting different kinds of formal verification would be appropriate.

Notice that by considering roles as aspects of objects in context of a particular task or activity, we are closer to the *functional approach* than to the *object approach*. In the functional approach an activity is described orthogonal to the objects involved in the activity, while in the object-approach objects are described orthogonal to the activities in which they are involved. Role modeling applies both these approaches at the level of design. However, every role is considered 'symmetric' to any other role concerning its communication capabilities. There is no explicit separation of symmetric and asymmetric, e.g client-server, relationships. Role modeling is used uniformly in both cases, and it may be considered a disadvantage that the distinction of such relationships are not made explicit. Believing that viewing the world as structures of interacting objects is 'natural' for many designs does not imply believing it for every (object-related) design.

2.2 Concepts as Role Models

We consider a role model to represent a concept in the domain of interest. Concepts usually comprise a structure and (often) a behaviour exerted on that structure. The structural aspect of a concept is denoted the *domain* of the concept. Hence the domain of a role model is the roles within the model. Structural relationships are represented by paths between the roles; each path representing a structural relationship. The domain of a particular role is the set of roles to which it has a path.

The behavioural aspect of a concept as a role model is represented by the messages sent to and from roles within the model. If there are no messages sent within a role model, it represents a purely structural concept. At present, the behavioural aspect of a concept is represented by interfaces of messages and the signatures of their corresponding methods; i.e the name, arguments and return types of the methods[3]. The arguments and return types are also specified by roles. The behaviour of each role comprises its interfaces towards its collaborators. Messages sent from the role are collected into separate *output interfaces* towards each collaborator, while messages to be received by the role are collected into a joint *input interface*. Hence, in addition to a graphical representation of a role model we explicitly specify the domain of the role model, and the domain and interfaces of each role.

2.3 An Example: A Bottle Deposit Machine

Before addressing the above issues in turn, we will first present a small toy example to be used to illustrate them. Consider the following description of a bottle deposit machine (BDM). The machine has a hole in the front leading to a conveyor belt. When a user arrives with some empty bottles, he inserts them into the hole onto the conveyor belt, one at the time. The machine has a laser equipment to check whether it is a bottle, whether it is in the 'right position' on the conveyor belt, and what its deposit value are. If it is not a recognized bottle the user gets an indication of 'no deposit', and the conveyor puts the bottle into a garbage can at the rear. If the bottle is in a 'wrong position', the user gets an indication of

[3] Assuming a message and its corresponding method to have the same name

'wrong position', and he removes the bottle from the conveyor belt. If the bottle is ok, the conveyor puts the bottle into a compressor, while it is indicated to the user to continue. On the front of the machine there is a light indicating whether the machine is full or not. If it is full, a user cannot insert any bottles until the machine is emptied. Finally the user can press a receipt button at any time. The machine then returns a receipt specifying the value of the bottles deposited.

From this brief description, assume we have three role models representing three different activities of the interaction between the BDM and the user. In this paper we assume the existence of these models, as it is outside the scope of this paper to present guidelines to which roles should be defined in a particular domain of interest. Nor do we present any guidelines as to which roles should be synthesized when composing roles and role models below. However, the CRC-method of responsibility-driven design described in [Beck] is well-suited to domain analysis for the purpose of selecting roles and establishing their mutual responsibilities in terms of messages sent and received.

First we have a role model representing the activity of the user depositing a bottle into the machine, and how the machine checks and deposits the bottle. This is illustrated in figure 1.

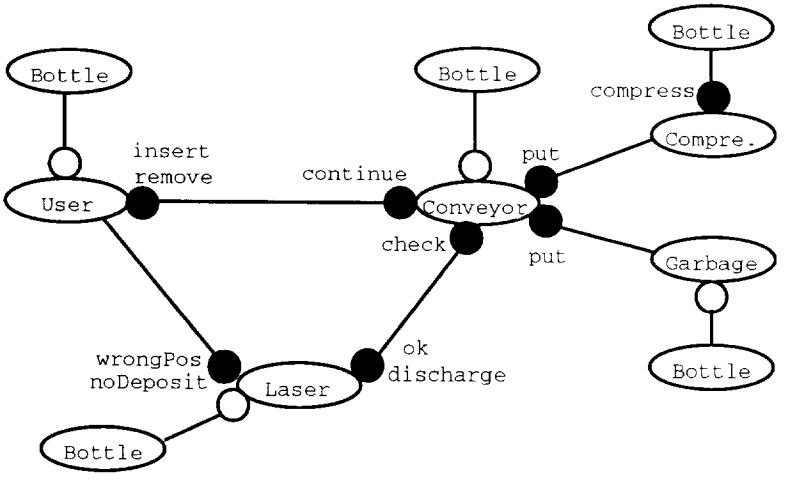

Figure1: Depositing a bottle

Then we have a role model representing how the user is indicated that the machine is full, or that it is emptied, illustrated by Figure 2.

Finally we have a model representing how the user requests and receives the receipt, as illustrated in figure 3.

The large ellipses represents the roles involved. In the model in figure 1 these are User, Bottle, Conveyor, Laser, Garbage and Compressor. Roles are connected by unidirectional or bidirectional *paths*, illustrated by lines. Roles connected by such paths are denoted *collaborators* of each other. Hence, the collaborators of User, denoted *Coll(User)*, is Bottle, Conveyor and Laser. The small circles at the end of some paths represent the *interfaces*

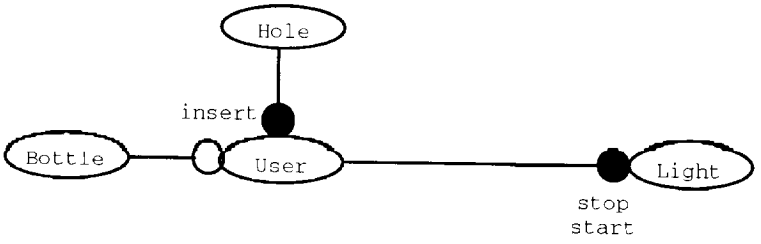

Figure2: The machine being full or emptied

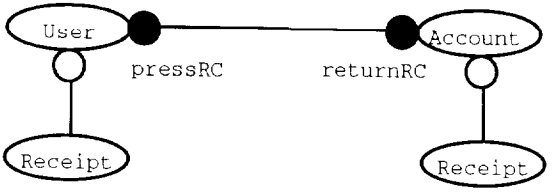

Figure3: Requesting and receiving a receipt

between the roles connected by this path. An interface is a collection of signatures for methods corresponding to messages sent and received over the path to which the interface is connected. The role to which an interface is connected is the role from which the messages in the interface are sent. Correspondingly, it contains the messages to be received by the role at the opposite end of the path. The role User may send the messages insert and remove to the role Conveyor, while it may receive the message continue from Conveyor. If the circle representing the interface is filled, then there are messages in the interface, while if it is not filled the interface is empty. If there is no interface along a path connected to a role, this role has no knowledge of the role at the other end of the path; no knowledge of a role in the sense of not having a reference to this role. Note however that not having a reference to a role does not imply not being able to receive messages from that role. If there is an interface connected to a role along a path, this role has a reference to the role at the other end of the path. Hence User will not send any messages to Bottle, but it has a reference to it. Since User has no interface towards Laser, it cannot send any messages to Laser or have any reference to Laser. However, User may receive the messages wrongPos and noDeposit from Laser.

Naming the role model in figure 1 DepositingBottle, its domain is as follows:

$$\mathsf{DepositingBottle} == \mathsf{User} \times \mathsf{Bottle} \times \mathsf{Conveyor} \times \mathsf{Laser} \times \mathsf{Compressor} \times \mathsf{Garbage}$$

On basis of the three role models above, the domain and interface descriptions of the three User-roles (the other roles are equivalent) are as follows:

```
User :: (Bottle × Conveyor)
        Out[Conveyor]:: insert    : Bottle→ .
                        remove    : .      → Bottle
        In            :: continue : .      → .
                        wrongPos  : .      → .
                        noDeposit : .      → .
```

User has a domain consisting of a reference to a Bottle and a Conveyor, while it has no reference to Laser. It has an output interface towards Conveyor consisting of the messages insert and remove. Their corresponding methods have the signatures of a Bottle as argument, and no return type, and no arguments but a Bottle as return type, respectively. The overall input interface of User is to receive the messages continue, wrongPos and noDeposit.

The domain and interface descriptions of the other roles named User (figure 2 and 3) are correspondingly:

```
User :: (Bottle × Hole)
        Out[Hole]:: insert    : Bottle→ .
        In        :: stop     : .      → .
                     start    : .      → .
```

```
User :: (Account × Receipt)
        Out[Account]:: pressRC    : .       → .
        In           :: returnRC  : Receipt→ .
```

3 Composing Role Models

3.1 Role Model Synthesis

If we have several role models representing different 'subconcepts' of a more aggregated concept, these subconcepts may be composed by *synthesizing* the corresponding role models. Synthesizing role models implies synthesizing one or more roles in each of the synthesized models, creating composite roles in a joint composite role model. Visually, synthesis can be considered as putting roles in different role models on top of each other in a pile, overlapping the roles to be synthesized. The result being a projection of this, as illustrated in figure 4.

Here three User roles are synthesized, creating a composite Customer role. This role has the same domain of collaborators and interfaces as the synthesized roles.

In general, the domain of the roles resulting from synthesis is the union of the domains of the synthesized roles. If role A and B are synthesized giving role AB, then if the domain of A is (X × Y) and the domain of B is (Z × W), the domain of AB becomes (X × Y × Z × W). Correspondingly, the set of output interfaces of AB is the set of output interfaces of both A and B, while the input interface of AB is the union of signatures in both the input interface of A and B. Furthermore, other roles having A or B in their domain now gets AB in their domain.

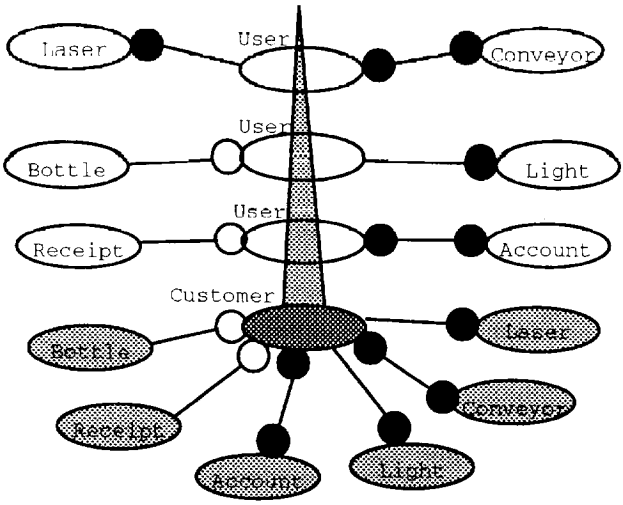

Figure4: Synthesizing role models

When synthesizing multiple roles from different role models, paths from synthesized roles to a common collaborator are composed into a single path. The interfaces in both ends of composed paths are also composed by composing the set of method signatures. Hence, synthesizing roles may imply synthesis of paths, which imply synthesis of interfaces.

Furthermore, the role models that are synthesized may contain messages that are the same common message in the resulting composite role model. Such duplicated messages will reside in the same composed interface after synthesis. Hence messages from different role models residing in the same interface may be merged by synthesis. Their signatures will probably correspond, but this is no requirement as they may be defined at different levels of abstraction.

Synthesis may lead to intended or casual name collisions. Casual name collisions are resolved by renaming. When doing synthesis, renaming may be applied without any constraints.

3.2 Composing our Exemplar Role Models

Above we have three role models representing different structural and behavioural aspects of the BDM and its interaction with a customer(User). Now we would like to compose these models into a single model representing the overall structure and behaviour of this system (to the brief level of detail considered in our toy example). This is achieved by synthesizing the role models, as illustrated in figure 5. Dotted lines illustrate synthesis, while the shaded roles are the composite roles in the resulting composite role model.

The synthesis is carried out by synthesizing the User roles in each of the models, the corresponding Bottle roles, and by synthesizing the Hole role in figure 2 and the Conveyor role in figure 1. The reason being that the User roles in all our models represent the same

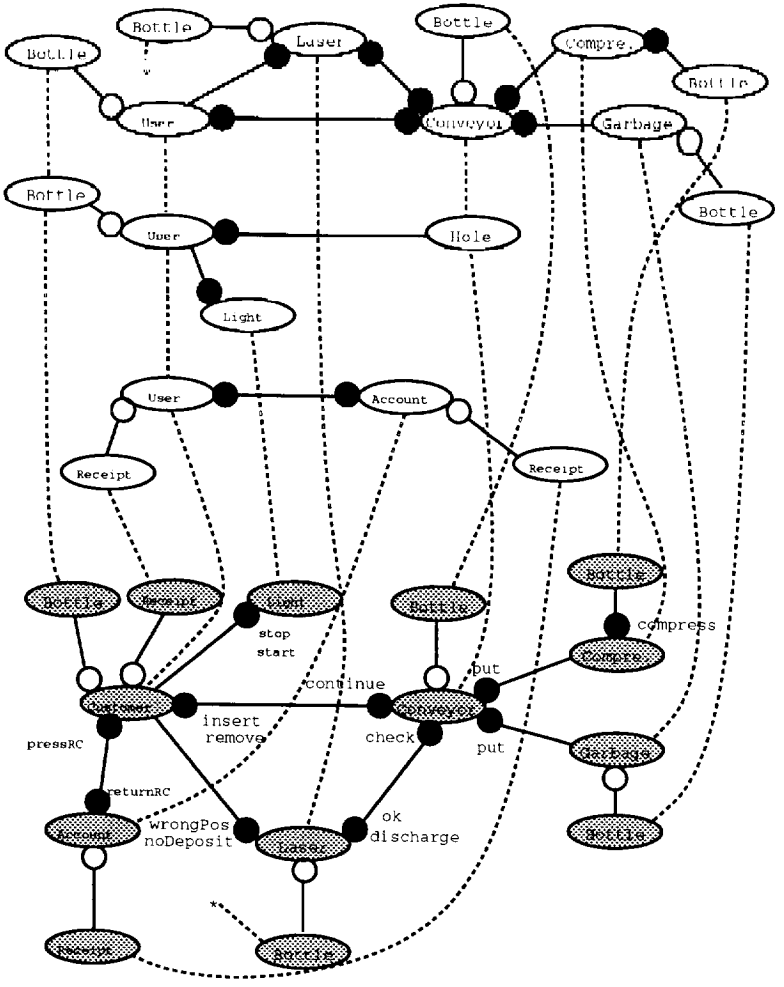

Figure5: Synthesizing the exemplar role models

real-world phenomenon, like a customer, and the Hole of figure 2 corresponds to the Conveyor of figure1 in the overall role model. The result of synthesizing the User roles is named Customer, while the result of synthesizing Conveyor and Hole is still named Conveyor.

When synthesizing Hole and Conveyor we realize that the insert messages in their input interfaces are the same. Hence they are merged by synthesis into a single message and corresponding method signature. This message is also denoted insert.

The domain of our composite role model of the BDM becomes:

$$BDM == (User \times Bottle \times Conveyor \times Laser \times Garbage \times Compressor \times Light \times Receipt \times Account)$$

The domain and interface description of Customer, resulting from the synthesis, is as follows:

```
Customer :: (Bottle × Conveyor × Account × Receipt)
        Out[Conveyor]:: insert    : Bottle  → .
                       remove     : .       → Bottle
        Out[Account]  :: pressRC  : .       → .
        In            :: continue : .       → .
                       wrongPos   : .       → .
                       noDeposit  : .       → .
                       stop       : .       → .
                       start      : .       → .
                       returnRC   : Receipt → .
```

We see that the common Bottle collaborators of the User roles are synthesized into a single Bottle collaborator. Furthermore, the Hole and Conveyor collaborators are synthesized. The insert messages are synthesized into a single message, also denoted insert.

The resulting domain and interface of the other roles in the composite role model are similar.

3.3 'Horizontal' Extension of a Role Model

Now that we have an overall but rather incomplete model of the BDM, we would like to extend it by introducing new structural and behavioural relationships.

A simple example would be to create a role model representing the relationship between Compressor and Light. If a client, not further described, puts a bottle into the compressor, the compressor may become full, not being able to compress any more bottles until it is emptied. Then compressor signals the light to indicate to its, not further described, client that no more bottles are to be deposited. This is illustrated in figure 6.

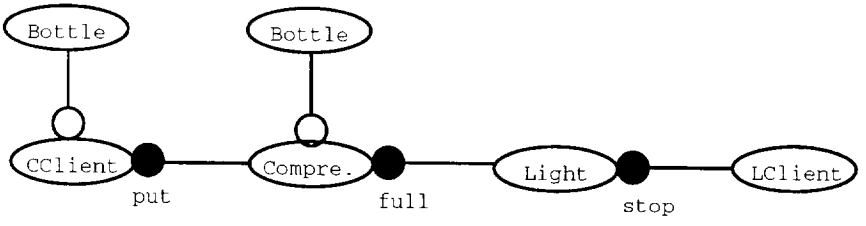

Figure6: Relationship of Compressor and Light

CClient put bottles into the compressor. Then the compressor may become full, in which case it sends a message full to Light. Light responds by the message stop to LClient representing a collaborator of Light.

This relationship is introduced into our overall role model by synthesizing both Compressor and Light roles, the corresponding Bottle roles, and by synthesizing CClient with Conveyor, and LClient with Customer. The put message of CClient towards Compressor is merged by synthesis with the put message of Conveyor towards Compressor. Fi-

nally the stop messages of Light are merged by synthesis. Figure 7 illustrates the result.

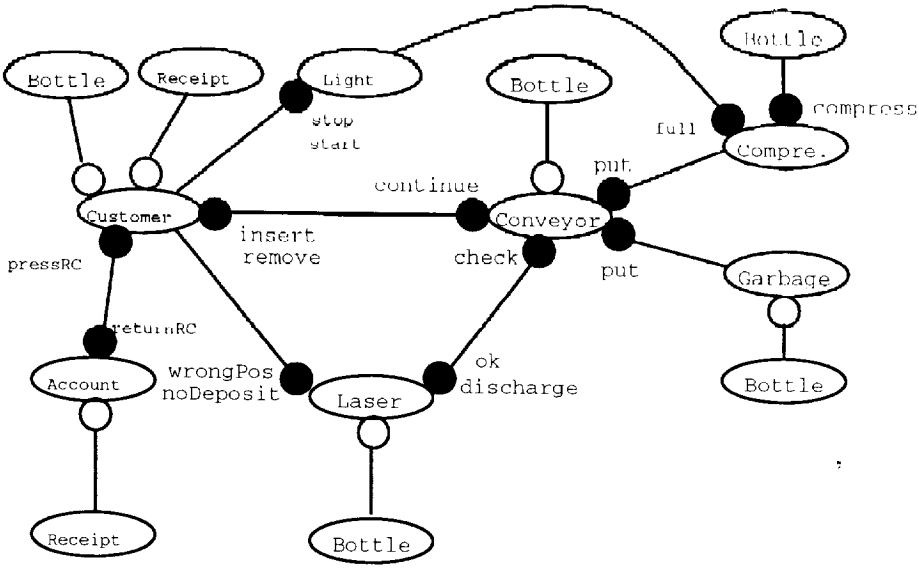

Figure7: Resulting overall model

Since the new structural and behavioural relationships introduced by this kind of synthesis are at the same level of detail as the original model, we denote it a *horizontal extension.*

3.4 Creating Abstractions

When synthesizing several role models we soon enter a morass of detail. Being able to create abstractions for hiding irrelevant details is important. This is achieved by creating *abstract roles.* An abstract role is like a virtual composite role representing an abstraction of a set of roles; denoted the *extension* of the abstraction. This extension can also be considered a role model within the role model. Notice the difference that despite every role being abstract in the sense of not being implemented, an abstract role represents a structure of hidden roles.

Abstract roles are created by a kind of *virtual synthesis*, similar to restriction in [Milner]. Virtual synthesis implies specifying the extension of the abstraction. These roles are synthesized into a composite role as described above, and every path and corresponding interface between the specified roles are hidden. However, only visually these roles are synthesized and their common paths and interfaces 'removed'. The abstract role does not exist except as a visual representative of its extension and their structural and behavioural relationships. The original role model is not really changed, hence the name virtual synthesis.

In our exemplar role model representing the BDM, we can specify an abstraction representing the machine itself, in opposition to the customer, receipt, and partly the bottles, existing outside the machine. This is specified by:

Machine == Conveyor × Laser × Garbage × Compressor × Light × Account

creating the abstract role denoted Machine. The overall role model is now as illustrated in figure 8. If starting design top-down, this would probably be our 'first' role model of the BDM.

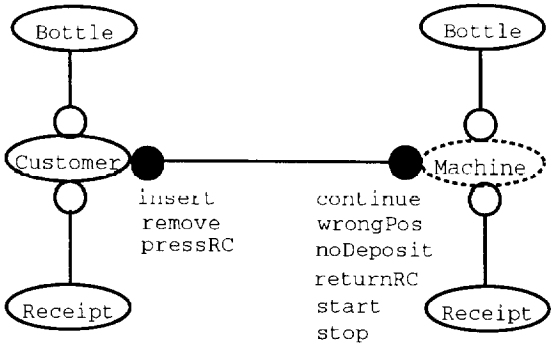

Figure8: Role model with an abstraction

The domain and interface description of Customer and Machine correspondingly becomes:

Customer :: (Bottle × Machine × Receipt)

Out[Machine]::	insert	: Bottle	→ .	
	remove	: .	→ Bottle	
	pressRC	: .	→ .	
In	:: continue	: .	→ .	
	wrongPos	: .	→ .	
	noDeposit	: .	→ .	
	stop	: .	→ .	
	start	: .	→ .	
	returnRC	: Receipt	→ .	

Machine :: (Bottle × Customer × Receipt)

Out[Customer]::	continue	: .	→ .
	wrongPos	: .	→ .
	noDeposit	: .	→ .
	stop	: .	→ .
	start	: .	→ .
	returnRC	: Receipt	→ .
In	:: insert	: Bottle	→ .
	remove	: .	→ Bottle
	pressRC	: .	→ .

When specifying the domain of a role, '==' is used to indicate that the role on the left hand side is an abstract role, having an extension comprised by the roles on the right hand side. '::' is used to indicate that the role on the left hand side has references, representing paths, to the roles on the right hand side. The domain of a role model is specified by using '==', since a role model can be considered an abstraction of the roles and the relationships within the model.

3.5 'Vertical' Extension of a Role Model

Assume we want to extend the BDM by e.g including a more detailed role model representing the internals of Laser. This cannot be achieved by horizontal extension as above. For this we apply a kind of *vertical extension*; extending the model by introducing more detail.

Assume the laser to consist of three internal modules. Ctr is a controller unit receiving requests to check a bottle. Chk is the unit doing the checking, while Ind is a unit returning the result of the check to two other units, denoted Ext and Int, outside the laser unit. Ext is a collaborator outside the machine, while Int is an internal collaborator. Figure 9 a) illustrates the more detailed laser model, not considering messages internal to Laser.

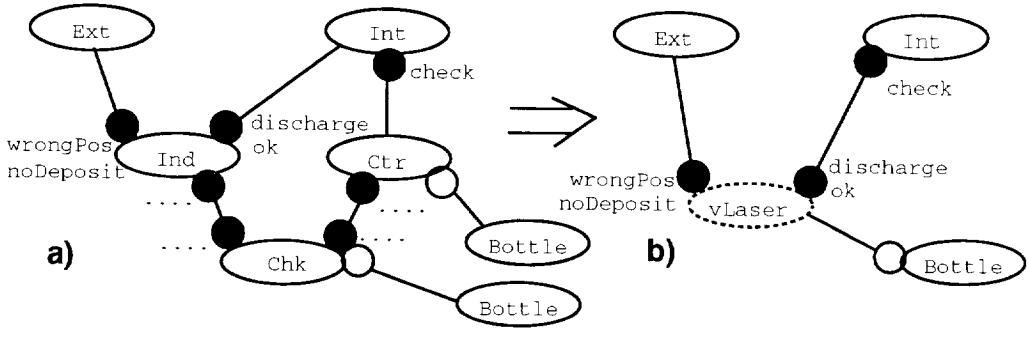

Figure9: Detailing Laser

To extend our overall role model, we first make an abstract role comprising the internals of Laser. This abstract role is denoted vLaser, as illustrated in figure 9 b). Then we *substitute* Laser by vLaser. The precise requirement of vLaser for doing such a substitution is described in section 5. However, we see that the Bottle collaborators are the same, the Ext collaborator corresponds to Customer, and the Int collaborator corresponds to Conveyor. Furthermore, vLaser have external paths and interfaces corresponding to every path and interface of Laser.

vLaser becomes an abstract role in the overall role model, comprising the internals of Laser. Part of the result of this vertical extension is illustrated in figure 10.

In general, a role model may be vertically extended by substituting roles with abstract roles comprising a structure of hidden roles. Such substitution makes the original role model

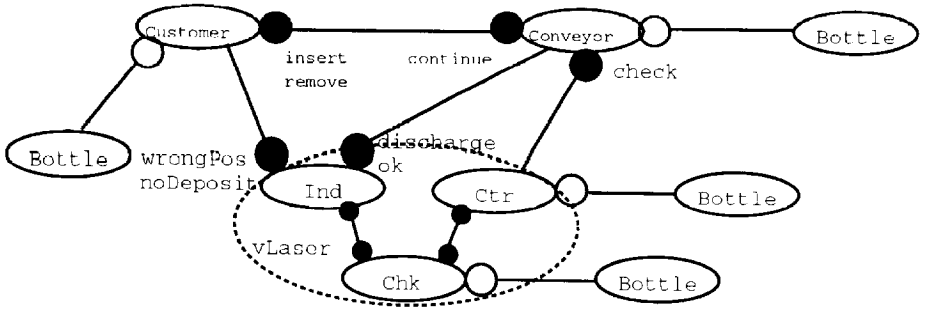

Figure10: Part of the overall role model after including internals of laser

more detailed. In case both the substituted role and the substituting role are abstract, several roles of the original role model is substituted by several other roles.

Note that substitution by abstract roles may imply splitting a path into several paths. An example being the path between Conveyor and the original Laser role. The original path may be considered an abstract path representing the paths between Conveyor and Ctr, and between Ind and Conveyor.

4 Design versus Implementation

4.1 Role Synthesis for Class Implementation

A role is considered an aspect or view of an object in a particular context. Hence roles are basically more fine-grained than the classes we implement to instantiate executable objects. A class is typically considered to represent some real-world phenomenon in our domain of interest. Hence several roles in several role models will correspond to the same real-world phenomenon to be implemented by a single class.

Synthesis may be used not only to compose designs, but also to synthesize individual roles from several role models corresponding to the same real-world phenomenon. This kind of synthesis is equivalent to synthesis of designs, as individual roles are synthesized in both cases. The resulting composite role may be used as a blueprint for implementing a class, from which objects may be instantiated that represents this phenomenon in all the contexts where it occurs.

An example of this is the synthesis of three User roles into a composite Customer role, as illustrated in figure 4. The Customer role corresponds to the same real-world phenomenon in several role models. By implementing a class corresponding to this role, we may instantiate objects configuring the User roles in the contexts in which they occur.

4.2 Synthesis of Design versus Implementation

So far when doing synthesis, we have only considered synthesis of designs. We have not considered having a possible 'underlying' implementation, in the sense of classes corre-

sponding to roles as in the section above. How do we relate the synthesis of designs to such an underlying implementation?

Synthesis of two or more role models implies the synthesis of two or more roles. We may consider the synthesis of multiple roles as a sequence of synthesis operations involving two roles at the time. Then there are three cases for synthesis, and one case for substitution of abstract roles.

Case 1 Synthesizing two roles both being a design only.

Synthesizing design is carried out as described above. This is typically the case when starting the development of a new application, before getting to the implementation phase. Usually some standard concepts are recognized early in the development, i.e a stack, a list, etc. Synthesizing role models representing these concepts usually involves their implementation. However, it is often the case that only some of the roles have an underlying implementation. A role model of the concept stack involves at least a role representing a user, or client, of the stack (sending messages like push and pop), a role representing the elements of the stack, and one or more roles representing the stack itself (responding to messages like push and pop). Typically the role representing the stack itself has an underlying implementation, while the client and element roles are design only; i.e stating requirements of objects configuring these roles. When synthesizing the role model representing a stack, usually these design roles are the roles to be synthesized, while the role(s) representing the stack itself is used as it is. Otherwise synthesis will be of case 2 or 3 below.

Case 2 Synthesizing a role already having an underlying implementation, as a class, and another role being a design only.

Synthesis of a role having a class C as its implementation, and another role representing a design only, can be realized by single inheritance. The resulting composite role may at some later stage be implemented by a class being a subclass SC of C. The extension of SC with respect to C will be features added by the synthesized design role. We may consider this synthesis a specialization of the role having an implementation.

This kind of synthesis is typical if we have a role model representing an existing, implemented application to be extended. First we make a separate role model representing a design of this extension. Then these role models are synthesized. The resulting role model will be partially implemented, and it may be fully implemented by extending the implementation of roles affected by the synthesized extension.

Case 3 Synthesizing two roles both having an underlying implementation.

How to realize the joint implementation of a composite role, synthesized from two roles both having an underlying implementation, primarily depends upon the language of implementation. The class representing the joint implementation can often be constructed by multiply inheriting the classes of the synthesized roles.

Synthesis of messages, and their corresponding methods, in the interfaces might correspond to the inheriting class redefining these methods to realize the effect of their merging. In case they are the same method, only one of their implementations should be inherited. Methods not synthesized are inherited side by side. Synthesis of paths as references is treated correspondingly. Name collisions have to be solved according to the language used, and renaming in the design is only possible if the language supports some renaming mechanism.

If the language of implementation does not support multiple inheritance, some restructuring of the inheritance hierarchy usually will be required, and such operations are usually not trivial. Furthermore, the direct correspondence between design and implementation is lost. Using a language supporting multiple inheritance usually eases the implementation of role models.

In general, manipulating implemented role models is limited by their language of implementation, while working on a design only implies more freedom of change. This is naturally so, as any implementation imposes constraints not existing in a design; being more concrete and concise inevitably implies fewer degrees of freedom when doing changes.

There are cases where synthesis of implemented roles may be handled by single inheritance. The class implementing one of them may be considered an abstract superclass of the class implementing the other one. An example being a role model representing the structure and behaviour of some abstract framework. Roles in this model may have some abstract classes as their implementation, while the roles in the synthesized model have a concrete implementation. When extended by the abstract implementation, it make them able to cooperate in the framework. Then the framework-classes could be abstract superclasses of the other concrete classes.

Case 4 Substitution of abstract roles involving implementations.

This is orthogonal to the three cases above. If we had an implementation of Laser in the example above, probably as an abstract class, and substituted it with the abstract role vLaser, what is the effect on our implementation?

Laser is replaced by Ctr, Chk and Ind. Hence the class Laser should either be deleted from the set of classes implementing our design, or if our language of implementation supports nesting of classes, the classes of Ctr, Chk and Ind could become nested classes within the class implementing Laser. Furthermore, references in other classes representing paths to Laser must be changed to represent paths to Ctr, Chk and Ind.

5 Configuring Role Models

5.1 Role-Type versus Object-Type

A role has a type with respect to each of its collaborators by the set of messages it can send to or receive from each collaborator. The domain and interfaces of a role defines a *role-type*. Such role-types specify operations applicable to an object by message sending, like an ordinary type, but it is not part of the type hierarchy for implemented types, i.e classes. Hence

role models provide a way of organizing types into reusable units without immediately tying them to the implementation of these types.

A role-type comprises what the collaborators expect of this role in context of the role model where it exists. The domain and interfaces of a role-type represents the minimum type requirement of an object to be able to configure this role. The requirement of an object to configure a role is that the type of the class from which the object is instantiated is *substitutable* [Wegner] with the corresponding role-type.

5.2 Type Conformance

To verify that the type of a class is substitutable with a particular role-type, we use the *type conformance relation* defined in [Cardelli, Canning], among others. This relation is based on comparing two interfaces, by comparing the signatures of the methods in the interfaces. If they satisfy the relation, in the sense that one of the interfaces *conforms* to the other interface, the type having the conformant interface may substitute the type of the other interface. This requirement assures that no type error can occur when substituting a type for another type; i.e no messages sent are not accepted by the receiver. Note that the message semantics considered here is sequential procedure calls.

The conformance relation for substitutability is defined as follows:

A type with interface S may substitute a type with interface T, if the interface S conforms to the interface T. An interface S conforms to an interface T, if every method (signature) in T have a conformant method (signature) in S.

A method m_s conforms to a method m_t iff:

1. m_s and m_t have the same name
2. m_s and m_t have the same number of arguments
3. the i'th argument type of m_t conforms to the i'th argument type of m_s
4. either none of them have any result type, or both of them have a result type, in which case the result type of m_s conforms to the result type of m_t.

Rule 3 implies *contravariance*, assuring that no messages are sent that are not accepted by the receiver. If S substitutes T, then if a collaborator of T sends a message corresponding to method m_t, the method m_s of S can be used instead if it satisfies the requirements of the collaborator concerning m_t. Hence m_s must accept at least all arguments accepted by m_t (rule 3), and the result of m_s must be acceptable in any context where the result type of m_t is expected (rule 4). A type hierarchy of substitutability based on contravariance is sound, and statically(compile time) checkable.

For practical purposes there are weaknesses by the contravariant rule. It is too strong, in the sense that it rejects interfaces of types which actually could substitute the other type without any errors. Many programming languages therefore allows *covariance*. By covariance, arguments in methods of a conformant type can conform to the arguments of the type to be substituted. Rule 3 then becomes 'the i'th argument type of m_s conforms to the i'th argument type of m_t'. Covariance provides more flexible, but not always sound, type hierarchies for substitutability. It is too weak, in the sense that it accepts substitutability of types

that may lead to errors. Such errors may be detected by tests during runtime [L-Madsen].

5.3 Object- and Role Configuration

The requirement of an object configuring a role is that the class of the object have an interface that conforms, by the relation above, to the input-interface of the role to be configured.

Depending upon how we constrain our implementation and configuration, we could have made the requirement that a configuring object only accept reception of a message if it comes from a collaborator as described in the role model. Then we would not collect every receivable message into the same input interface, but have one input interface towards each collaborator. An example being that e.g Conveyor only accepts insert from the object configuring the Customer role, not from any other collaborator.

There are in principle no requirements of output from the configuring object. However, collaborators of the configured role only satisfy (at least) those requirements made in the role model concerning output interfaces from its collaborators. Hence the configuring object must not send any other messages to a certain collaborator than those described in the output interface of this collaborator in the configured role.

Furthermore, the object is required to have a set of references corresponding to those in the domain of the role to be configured.

Substitution of abstract roles corresponds to configuring the abstract role to the role that is to be vertically expanded. In the example in section 3.5, the abstract role vLaser configures Laser. In a sense we have *role configuration* and *object configuration* as two similar cases of configuration. When doing role configuration we substitute a role by another role. As for objects, we should check that the type of the configuring role conforms to the type of the configured role.

5.4 Constraints on Configuration

When configuring roles in a role model, there are certain constraints to be satisfied. Constraints on configuration are basically of two kinds:

- Several roles in the same role model together add up the requirements of objects configuring these roles
- Some roles cannot be configured by the same object

Hence it is usually not correct freely to configure separate roles in the same role model. It has to be specified separately which roles are configurable, interdependencies of their configurability, and their overall type requirements.

A typical example of the first kind of constraint, is that roles representing elements to be inserted into a list also have to satisfy the requirements of roles representing elements within the list, as well as roles representing elements removed from the list (they will sooner or later be played by the same object). Hence, the type requirements of objects configuring these roles corresponds to type requirements of a composite role created by synthesizing these roles.

Another example is the role Bottle in our example above. The bottle inserted into the machine is the same bottle that is later inspected by the laser. Therefore the object configuring Bottle as a collaborator of User has to satisfy requirements made by Laser (not included in our simplified model). Later on, the object configuring Bottle is either passed on to Garbage or to Compressor. Hence no single object can configure both of these Bottle roles, becoming a collaborator of both Compressor and Garbage. An object configuring Bottle may either satisfy requirements of Garbage, or of Compressor. In this particular case that depends upon its behaviour towards Laser. Laser checks whether it is a bottle to be deposited and compressed, or some other thing not legally deposited and therefore to be sent to Garbage. The requirement of the object configuring Bottle is that if it is considered legal by Laser it has to satisfy requirements made by Compressor, otherwise it has to satisfy requirements made by Garbage.

Such interdependencies may be specified by:

$$\text{Bottle} = Bottle_{User} \times Bottle_{Laser} \times Bottle_{Conveyor} \times (Bottle_{Compressor} + Bottle_{Garbage})$$

Stating that an object configuring Bottle either has to satisfy the type requirements of the composite role synthesized from Bottle collaborating with User, Laser, Conveyor and Compressor, *or* (exclusive) the composite role synthesized from Bottle collaborating with User, Laser, Conveyor and Garbage. In addition the designer has to assure that only the role $Bottle_{User}$ is initially configurable. The designer also has to assure that objects configuring bottles collaborating with Garbage, not with Compressor, will be rejected as illegally deposited bottles by Laser.

6 Summary

We believe that the synthesis technique is useful for supporting several key issues in object-oriented design. Separation of concern is achieved by first deriving several independent role models, possibly at different levels of detail. These models may later be composed, and irrelevant details hidden by creating abstractions. Role models may be extended horizontally by other role models at the same level of detail, or vertically by role models at different levels of detail. This supports a flexible design approach. The designer may start doing preliminary design at any level of detail, without taking the entire system design into consideration; i.e either start top-down, bottom-up, or at some intermediate level. Different parts of the same overall design may be derived at any of these levels independent of other parts of the same design. Concerning the relation of aggregation versus specialization, an aggregation is considered a specialization if the extension by synthesis is comparable to the extension of a superclass by the increment of a subclass. Hence, the same synthesis operation may be considered an aggregation and/or specialization depending on how, and from where, the result is viewed. By synthesizing individual roles in several role models we get the type requirements of classes implementing our design. Furthermore, a role model may be considered a classification of structures of executing objects having a structure and behaviour corresponding to the role model. Finally, a role model design may be reused either by synthesizing it with other role models, or by configuring its roles. The ability to synthesize any role model, and the ability to configure any role in any role model (with a few exceptions concerning constraints), provides a basis for reuse of designs.

7 Further Work

Here we have considered the interface of a role towards its collaborators as a set of signatures only, the semantics of message passing being sequential procedure calls. Furthermore, our models are abstract having limited expressibility concerning details. The sending and reception of messages is related and interdependent. There are temporal relations between these as described in [Arapis]. Currently we are investigating the possibility of describing the behaviour required of a role as a labeled transition system. Each role will have a behaviour protocol containing its abstract states and transitions. A transition either being a message sent, received, or an internal transition invisible to its collaborators. The message semantics we are considering are synchronous and asynchronous concurrent, in addition to the case above. The decision of whether a role or an object is able to configure a particular role should then be based upon some type conformance relation of labeled transition systems. An attempt to establish such a relation, denoted *interaction conformance*, is discussed in [Nierstrasz91]. This work is related to the work on CCS (Calculus of Communicating Systems) by [Milner], among others. We are working on establishing such a type conformance relation for different kinds of message semantics.

In section 4 we are not being very accurate concerning differences in the programming languages in which classes corresponding to roles are implemented. Hence an issue is to provide more accurate guidelines for how composition of the domain and behaviour of roles in a design is reflected in a possible underlying implementation.

Acknowledgments. This work is supported by The Norwegian Research Council for Science and the Humanities(NAVF). We especially would like to thank Else Nordhagen[Nordhagen], and also Arne Jørgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Erik Næss-Ulseth, Gro Oftedal, Grete Christina Olsen, Anne Lise Skaar and Pål Stenslet as major contributors to the development of the role modeling technique. We are most grateful to Vladimir Bacvanski, Ralph Jungclaus, Stein Krogdahl, Pieter Jan Morssink, Jan Overbeck and the anonymous referees for their valuable comments on drafts of this paper.

8 References

[Arapis] C.Arapis, *Specifying Object Life-Cycles,* Ed.D.Tsichritzis, Object Composition, Universite de Geneve, 1990, pp.197-225

[Beck] K.Beck, W.Cunningham, *A Laboratory for Thinking Object-Oriented Thinking,* Proc. of OOPSLA '89; Object-Oriented Programming Systems, Languages and Applications, October 1989

[Canning] P.S.Canning, W.R.Cook, W.L.Hill, W.G.Olthoff, *Interfaces for Strongly-Typed Object-Oriented Programming,* Proc. of OOPSLA '89; Object-Oriented Programming Systems, Languages and Applications, October 1989, pp.457-467

[Cardelli] L.Cardelli, *A Semantics of Multiple Inheritance,* Semantics of Data Types, LNCS 173, Springer-Verlag, 1984, pp.51-67

[**Champeaux**] D.de Champeaux, *Object-Oriented Analysis and Top-Down Software Development,* Proc. of ECOOP '91; European Conference on Object- Oriented Programming, Geneva, Switzerland, July 1991, pp.360-376

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

[**Helm**] R.Helm, I.M.Holland, D.Gangopadhyay, *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems,* Proc. of ECOOP/OOPSLA '90; European Conference on Object- Oriented Programming/Object-Oriented Programming Systems, Languages and Applications, Ottawa, Canada, October 1990

[**Kilian**] M.F.Kilian, *A Note on Type Composition and Reusability,* OOPS Messenger, Vol.2, No.3, July 1991, pp.24-32

[**L-Madsen**] O.Lehrman Madsen, B.Magnusson, B.Møller-Pedersen, *Strong Typing of Object-Oriented Languages Revisited,* Proc. of ECOOP/OOPSLA '90; European Conference on Object- Oriented Programming/Object-Oriented Programming Systems, Languages and Applications, Ottawa, Canada, October 1990

[**Meyer**] B.Meyer, *Object-Oriented Software Construction,* Prentice-Hall, 1987

[**Milner**] R.Milner, *Communication and Concurrency,* Prentice Hall, 1987

[**Nierstrasz90**] O.Nierstrasz, M.Papathomas, *Viewing Objects as Patterns of Communicating Agents,* Proc. of ECOOP/OOPSLA '90; European Conference on Object- Oriented Programming/Object-Oriented Programming Systems, Languages and Applications, Ottawa, Canada, October 1990, pp.38-43

[**Nierstrasz91**] O.Nierstrasz, M.Papathomas, *Towards a Type Theory for Active Objects,* OOPS Messenger, Vol.2, No.2, April 1991, pp.89-93

[**Nordhagen**] E.Nordhagen, *Generic Object-Oriented Systems,* Proc. of TOOLS '89; Conference on Technology of Object-Oriented Languages and Systems, Paris, Nov.1989, pp.131-140

[**Pernici**] B.Pernici, *Objects with Roles,* Proc. of the Conference on Office Information Systems(COIS), Cambridge, Massachusetts, 1990

[**Reenskaug1**] T.Reenskaug, E.Nordhagen, *The Design and Description of Complex, Object-Oriented Systems, Ver.1.0,* Center for Industrial Research, Report no.89 272-1,Nov.1989

[**Reenskaug2**] T.Reenskaug, E.P.Andersen, A.J.Berre, A.Hurlen, A.Landmark, O.A.Lehne, E.Nordhagen, E.Næss-Ulseth, G.Oftedal, A.L.Skaar, P.Stenslet, *Seamless Support for the Creation and Maintenance of Object Oriented Systems,* To appear in JOOP, Journal of Object-Oriented Programming

[**Shilling**] J.J.Shilling, P.F.Sweeney, *Three Steps to Views: Extending the Object-Oriented Paradigm,* Proc. of OOPSLA '89; Object-Oriented Programming Systems, Languages and Applications, October 1989, pp.353-361

[**Wegner**] P.Wegner, S.B.Zdonik, *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like,* Proc. of ECOOP '88; European Conference on Object- Oriented Programming, Oslo, Norway, August 1988, pp.55-77

[**Wirfs-Brock89**] R.J.Wirfs-Brock, B.Wilkerson, *Object-Oriented Design: A Responsibility-Driven Approach,* Proc. of OOPSLA '89; Object-Oriented Programming Systems, Languages and Applications, October 1989, pp.71-75

[**Wirfs-Brock90**] R.J.Wirfs-Brock, R.E.Johnson, *A Survey of Current Research in Object-Oriented Design,* Communications of the ACM, September 1990, vol.33, no.9, pp.105-124