The OOram Meta-Model

combining
role models, interfaces, and classes
to support
system centric and program centric modeling

A proposal in response to OMG OA&D RFP-1 submitted by

Taskon A/S
Reich Technologies
Humans and Technology

8 January 1997

Version 1.0 of 8 January 1997

The submitters

Taskon
Gaustadalléen 21, N-0371 Oslo 3 Norway. Tel. + (47) 22 95 86 31
Fax: + (47) 22 60 44 27
http://www.sn.no/taskon/

Reich Technologies US
Cecilia Schuster, VP of Operations
PTEC 209, 19E Central Ave.
Paoli, PA, 19301, USA
Tel.: +(1) 610 889 9606 Fax: +(1) 610 640 0619
e-mail: 100437.3555@compuserve.com
http://www.projexion.com.

Reich Technologies Europe
Georges-Pierre Reich, Chief Architect [100431.124@compuserve.com]
Agnes Guichard, International Public Relations
[106003.2514@compuserve.com]
40 Quai de la Douane
29200 Brest - France
Tel.: +(33) 2 98 80 48 99 Fax: +(33) 2 98 44 77 72
http://www.projexion.com.

Humans and Technology
7691 Dell Rd., Salt Lake City, UT 84121, U.S.A. Tel +(1) 801 943 8484 Fax
+(1) 801 943 8499
http//www.projexion.com

Contributors in alphabetical order

Jan-Øyvind Aagedal, SINTEF [Jan.Aagedal@informatics.sintef.no]
Arne-Jørgen Berre, SINTEF [Arne.J.Berre@informatics.sintef.no]
Alistair Cockburn, Humans and Technology [arc@acm.org]
Jon Oldevik, SINTEF [Jon.Oldevik@informatics.sintef.no]
Trygve Reenskaug, Taskon (editor) [trygve@taskon.no]
Georges-Pierre Reich, Reich Technologies [100431.124@compuserve.com]
Rebecca Wirfs-Brock, ParcPlace-Digitalk [rebecca@parcplace.com]

Table of contents and index at the end of the document

Summary

The OOram Object Analysis & Design Metamodel describes fully integrated models of object systems. The models seamlessly combine the expressiveness of interface abstractions, conventional class abstractions, and powerful system abstractions on structures of interacting objects.

The main contributions of this proposal are its overall architecture and its system abstraction. The architecture is object oriented and unifies a number of powerful abstractions. The system abstraction combines the powers of use cases, responsibility driven design, and role modeling; it can be thought of as an extension of the UML and OML object models.

We propose adding three basic concepts to the object models: *role model, role,* and *port.* These abstractions, together with the existing OMG interface abstraction, offer several advantages to OMG. The most important are:

1. They support the description of systems of interacting, distributed objects without binding program specifications such as types and classes.
2. They support the separation of concern, permitting the modeling of an *Area Of Concern* such as a subject or a use case. The model shows the static and dynamic properties of a structure of collaborating objects, where each object is represented by a partial description covering relevant aspects only.
3. They support the description and reuse of generic patterns of objects, permitting the standardization of the static and dynamic properties of object systems. Users of these descriptions can specialize them, adding new functionality while retaining their inherent static and dynamic correctness.

The models described by the submitted meta-model are pure object models that can be stored in the network and accessed through CORBA.

As far as is known at the time of writing, the concepts of the OOram meta-model are compatible with the other proposals and can be merged with them. The notation presented in this proposal is tentative and can be modified to conform to any preferred standard.

Chapter 1
The Proposal

1.1 Summary of Main Ideas

This proposal is based on three fundamental ideas:

1. *An object-oriented meta-model.* The OMG is all about enabling object technology to support distributed integrated systems. Our proposed **meta-model** is object oriented; models can therefore be realized as OMG object systems that describe other object systems. Further, since the meta-model is just another object system, the meta-model itself can basically be described in its own terms.
2. *The role and class dichotomy.* We propose two orthogonal and complementary approaches to object modeling. They focus on different aspects and are useful for different purposes in the system life cycle.
   - The *System-Centered approach (SC)* focuses on systems of interacting objects. Use cases, Responsibility Driven Design, collaboration diagrams, message scenarios, and system behavior models belong in this dimension. A precisely defined abstraction is the *role model*, which describes the static and dynamic properties of a structure of interacting objects in the context of a specific subject, called an *Area of Concern.*
   - The *Program-centered approach (PC)* focuses on the class as the central abstraction. Supporting concepts are types and various kinds of relationships and associations between classes and types.
3. *Flexible model requirements.* System models are made available on an 'as needed' basis. Model parts are defined so that they can be included or excluded as needed. An actual model need only contain information considered useful for its intended purposes. The meta-model is defined so that consistency is ensured for all included parts. A *hierarchy of model parts* describes mandatory priorities.

Some of the RFP-1 Requirements are met as follows:

1. *OA&D Interoperability (p.4).* A client of the proposed models can navigate among their objects and retrieve their information. OA&D tools can thus extract information generated by other tools, and similarly make their repositories available to other tools. OMG can publish generic models as object structures in a generally available object repository. The provider of an application can publish its application model, enabling its customers to adapt to the model.
2. *Proposal shall not constrain processes and techniques of OA&D (p.4)* The proposed meta-model neither describes nor restricts processes and techniques.

3. *Submissions must define a meta-model that represents the semantics of OA&D methods, including one or more of the following core models: Static models, dynamic models, usage models, architectural models. (p. 3, 15, 18)* The proposed meta-model includes the basic aspects of the core models, and can easily be extended to cover other aspects. We find static models in both dimensions. Dynamic models, usage models, and architectural models all belong to the system dimension.
4. *Define, explain, and/or demonstrate integration with future OMG specifications. (p.13).* OMG standardizes distributed, interacting objects. Role modeling is an abstraction on interacting objects that extends the expressibility of the interface from a binary, uni-directional construct to multi-object and multi-path constructs. Role modeling thus provides a powerful tool for future OMG standardization.
5. *Submission shall be OO and expressed in OMG IDL (p.10).* The submitted meta-model is a pure object model, all interfaces are expressed in IDL.
6. *Different OMG specifications must be able to work together. (p.12).* The proposed meta-model merges well with other OMG specifications. The intentions of the standardizers could be clarified by adding role models to some of the existing standards. (E.g., the Common Object Services Specification).
7. *Specifications should be extensible through an iterative process. (p.12).* The specified meta-model is open ended, and can easily be extended. Indeed, the proposed role model subsystem is a subset of the meta-model supported by the current OOram tool products. It is also a subset of the meta-model described in *Working with objects* [Ree-96].
8. *It should be possible to define and standardize new specifications without having to redesign existing systems using existing OMG specs. (p.12).* There are no conflicts between the proposed meta-model and existing OMG standards.
9. *It should be possible to configure OMG specs (p.12).* The meta-model is designed such that its elements can be included or excluded as needed by the users.
10. *Integrity, Reliability, Safety, Performance, Scaleability, Portability (p.12).* The proposed meta-model is internally consistent. Existing tools check and/or enforce this model consistency.

1.1.1 An object-oriented meta-model

*"The Object Management Group's central mission is to establish an architecture and set of specifications, based on commercially available object technology, to enable **distributed integrated applications**. Primary goals are **reusability, portability,** and **interoperability** of object-based software components in distributed heterogeneous environments."* [RFP-1]

Figure 1 illustrates that the essence of the OMG idea is a world of interacting objects. The common object reference model hides differences in communication networks and variations in

platform hardware and software. The resulting system of objects is "pure" in the sense that its components are conceptually simple and well-defined.

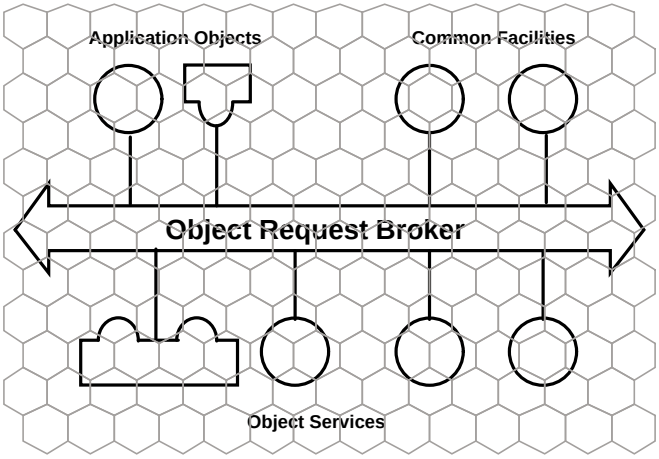

Figure 1. The world of OMG consists of interacting objects

Practical object structures will be large and complex; potentially encompassing a global information network. Nobody will know it all; it is inconceivable that we should be able to create a global model of the whole of Internet. What we can do is to select any part of it as our system of interest, and create a model of that system. Our definition of *system* is taken from [Ree 96]. It permits us to choose various phenomena of interest and to study each of them to any desired degree of precision:

> A **system** is a part of the real world which we *choose* to regard as a whole, separated from the rest of the world during some period of consideration; a whole that we *choose* to consider as containing a collection of objects, each object characterized by a selected set of associated attributes and by actions which may involve itself and other objects.

The systems will typically be *open systems;* i.e., systems that interact with their environment:

> For a given system, the **environment** is the set of all objects outside the system whose actions affect it, and also those objects outside the system whose attributes are changed by its actions.

Ultimately, every object in the world may either directly or indirectly be dependent upon every other object. We *choose* a system that is meaningful and useful in a given context. We can choose an application, an object service, or a common facility as the system of interest. An enterprise may, for example, elect to focus on a system of objects that represents its current project plans. A bank may choose a system of objects that represents the bank as seen by a customer. A systems vendor may decide to focus on an event notification service.

A system of objects is usually changing over time, and it will generally be too large and complex to be comprehended as a whole with all its details. As illustrated in figure 2, we create a *model* of the system that highlights aspects of interest and ignores everything else:

1. *A model is created for a purpose.* A good model serves its purpose well, a bad one does not.
2. *A model is never complete.* An analysis model ignores design details. A design model ignores code details. Program code hides the details of library functions and hardware properties. And all models ignore phenomena outside their area of concern.
3. *We think in multiple, often overlapping, models.* Our models reflect our purposes. When we change focus of interest, we also switch to a different model.

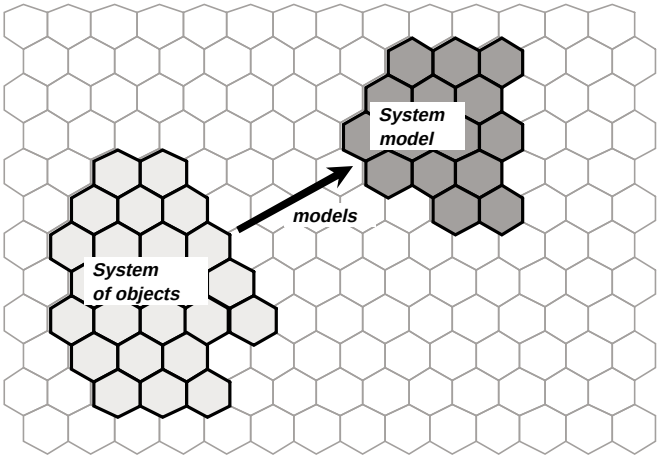

Figure 2. A system and its model

Models can be *generic*, describing the properties common to a number of systems. So the OMG Business Object Modeling Task Force can standardize on a generic model of a purchasing transaction. In each of its specializations, it can be elaborated with the details that are internal to particular enterprises.

The OA&D RFP-1 requests proposals for standardizing common properties of system models. We *choose* a generic system model as our system, we *choose* a purpose such as interoperability or business object standardization, and we create a model of the model -- called a **meta-model** as illustrated in figure 3.

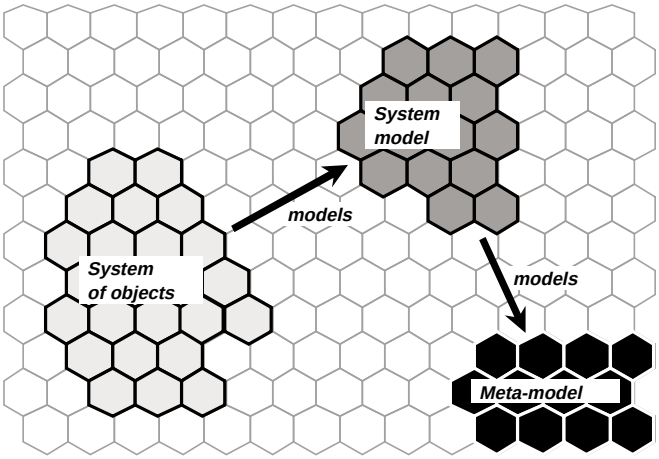

Figure 3. The meta-model

This object-oriented meta-model is the subject of this proposal.

1.1.2 The role and class dichotomy

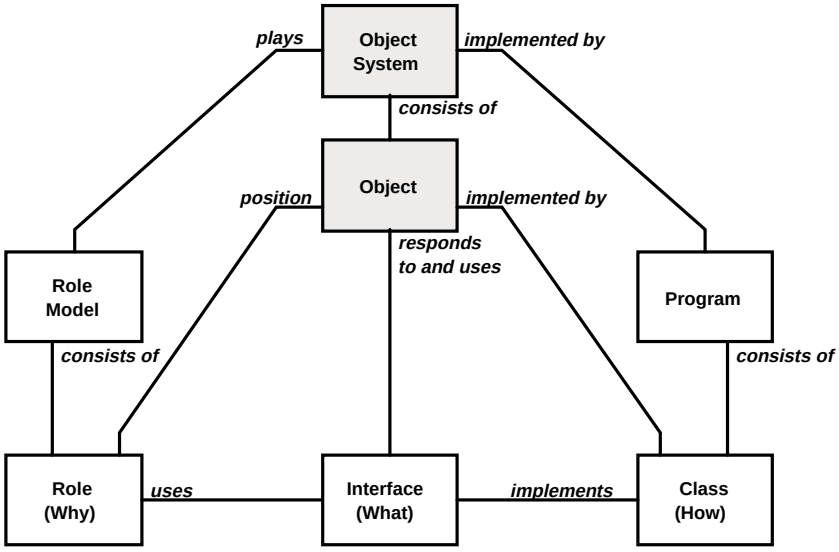

Figure 4. Semantic relationships between the object system and its major abstractions

We propose two approaches to object modeling: the *System-Centered approach (SC)* and the *Program-centered approach (PC)*. SC provides abstractions on systems of interacting objects. PC provides abstractions on the classes of object-oriented programs and their relationships.

The two approaches are essentially orthogonal. Their main point of contact is through message interfaces: Interfaces describe the system interactions in the SC approach, and they describe the class capabilities in the PC approach. We will discuss the two approaches under the following headings:

1. *System specification and Use Cases.* A system is described as seen from its environment. We see the actors as system environment objects, we see their goals, and we see the messages that they use to trigger system functionality. Use cases are essentially system oriented, and belong entirely to the system dimension.
2. *Roles versus classes.* A *role* is an abstraction on an object's position in a system of interacting roles. A *class* is an abstraction on the properties of a set of similar objects (i.e., the instances of the class).
3. *The dynamics of objects and systems.* The dynamic properties of an isolated class can be described in the PC approach. One form of description is a Finite State Machine associated with the class. The dynamic properties of a system cannot be expressed in the PC approach, since the class does not represent the object identity property.

   The dynamic properties of a system are precisely expressed in the SC approach. A *scenario* shows a sample message interaction sequence, while a coordinated set of *Finite State Machines* can show the behavior of the system and all its parts.
4. *Interaction versus service.* A role model identifies both the sender and the receiver of message interactions. In a class model, the focus is on services offered and the message interface of a class is identified. In the system dimension, behavior is thus described in terms of message interactions, each message having a sender and a receiver. In the program dimension, behavior is described in terms of the services offered by the class to its environment, each received message being matched by a **method** that handles the message. The message *interfaces* bridge the two dimensions, because the messages transmitted in an interaction must be understood by its receiver.
5. *Class inheritance versus system inheritance.* Class inheritance provides the inheritance of object state and behavior. System inheritance additionally supports the inheritance of system states and overall system behavior, given added leverage to object patterns and frameworks.

1.1.2.1 System specification and Use Cases

A **use case** describes the responsibilities and externally visible actions of the system under design and the actors in its environment, in pursuing a goal of an actor in the environment. A use case can have one of several outcomes: the actor's goal is delivered, partially delivered, or abandoned.

A use case consists of:

1. the goal statement
2. operating conditions (conditions under which the use case happens)

3. one or more scenarios

As an example, consider the *activity networks* used in project planning an control. An activity is an abstraction on a task. It is characterized by having a duration, a number of predecessor activities, and a number of successor activities. An activity can start when all its predecessors are completed. Its earliest finish is its duration after the finish of the latest predecessor. And its successors can start any time after its finish time. The earliest start of the network activities can be easily computed by a *front-loading* operation: Start with the project start time and the initial activities, add activity durations, and iterate through the activity successors. Similarly, a *back-loading* operation computes the latest completion time for all activities starting from the end activities and the project completion time. Figure 5 shows an illustrative example of a network together with a chart showing earliest and latest activity times.

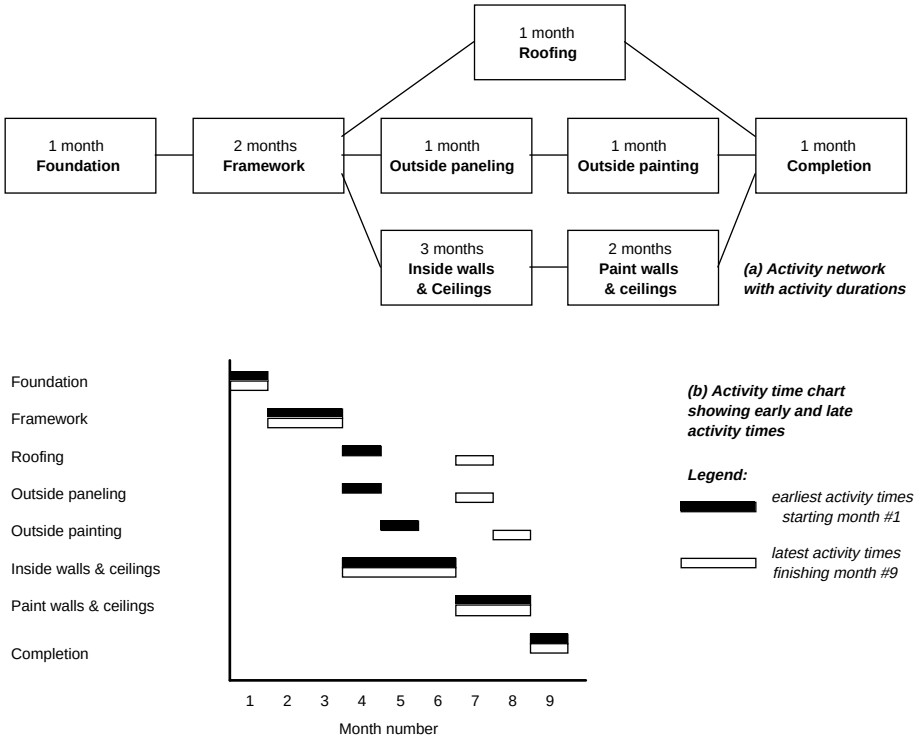

Figure 5. Example activity network

Figure 6 shows the system together with scenarios for the goals of computing the earliest, resp. latest, activity times.

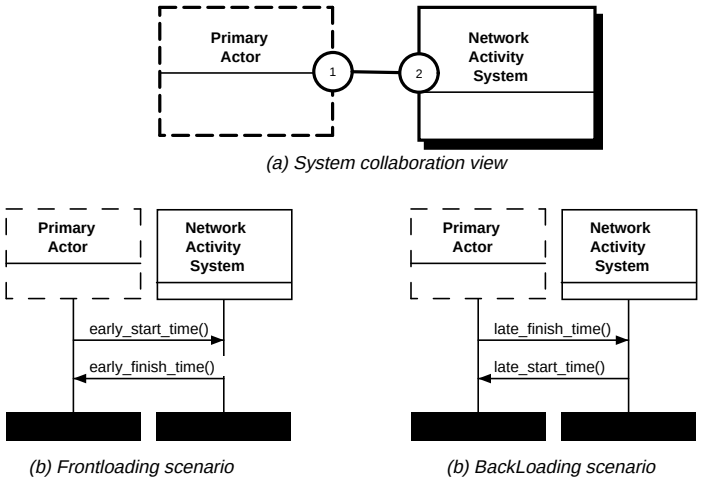

Figure 6. Project planning system seen from its environment

1.1.2.2 Roles versus classes

Role models are strong on system properties, weak on object properties. Class models are strong on object properties, weak on system properties.

In an object-oriented implementation of an activity network, it seems natural to represent each activity as an object and to let the front- and back-loading operations be performed by this object structure. Figure 7 shows the PC and SC approaches to this solution.

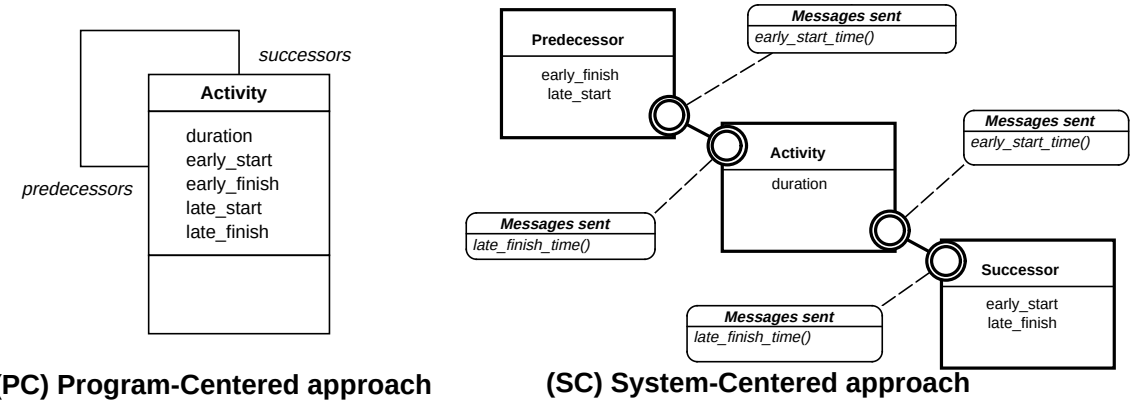

Figure 7. Vanilla network activities

The role model to the right in figure 7 is read as follows: The system under consideration

consists of a number of objects. The role model shows three archetypical objects representing an Activity, one of its Predecessors, and one of its Successors. An object playing the Predecessor role supports the attributes *early_finish* and *late_start*. An object playing the Activity role knows the *duration* attribute, and an object playing the Successor role knows the *early_start* and *late_finish* attributes.

The small circles represent object references: there are two-way references between Predecessor and Activity, and between Activity and Successor. The small circles are here doubled to indicate that these are in many-relations; e.g., an Activity object knows any number of predecessors where the Predecessor role represents one, archetypical instance.

All roles can be implemented by the same class as indicated to the left in figure 7. The predecessor-successor relationships is indicated in the diagram.

There is a subtle, but very important, difference in the semantics of the two models. The relation concept in the class model is similar to the relation concept in E-R models. Its meaning is that an instance of the Activity class has a predecessor relationship to some instances of the same class, and similarly for the successor relationship. There is no *equivalence of path*, however; the predecessor of an activity need not necessarily have that activity as its successor.

In contrast, the role model retains object identity. In an instance of a role model, each role represents one specific object. So in the role model of figure 7, the system objects are linked exactly as shown; the role model is a faithful representation of an object pattern. This is the key to the role model's representation of system as a whole with all its static and dynamic properties.

1.1.2.3 The dynamics of objects and object systems

The dynamic properties of single objects can be described in the PC approach. One form of description is a Finite State Machine associated with a class. The dynamic properties of a whole system cannot be expressed in the PC approach, since the class does not reflect the object identity property.

The dynamic properties of a system can be precisely expressed in the SC approach. A *scenario* shows a sample message interaction sequence, while a coordinated set of Finite State Diagrams can show the behavior of the system and all its parts.

A frontloading operation is shown in figure 8. The scenario view in (a) shows the typical message interaction sequence. The method view in (b) shows the method performed by an Activity object when receiving the *early_start_time* message from a predecessors. The internal operations are shown in pseudo-code, while the receiving and sending of messages is shown explicitly since they are essential to understanding the system as a whole.

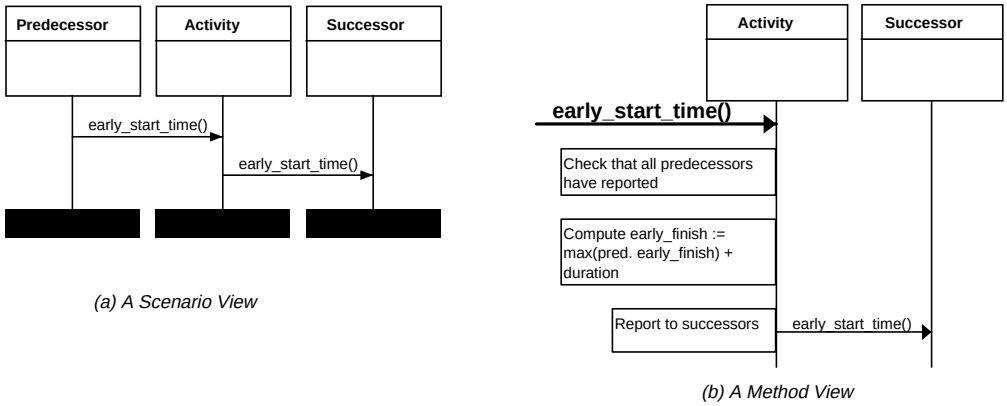

Figure 8. Simple frontloading computation

Similarly, in figure 9 we show how late start times are computed by activities receiving the *late_finish_time* messages from their successors.

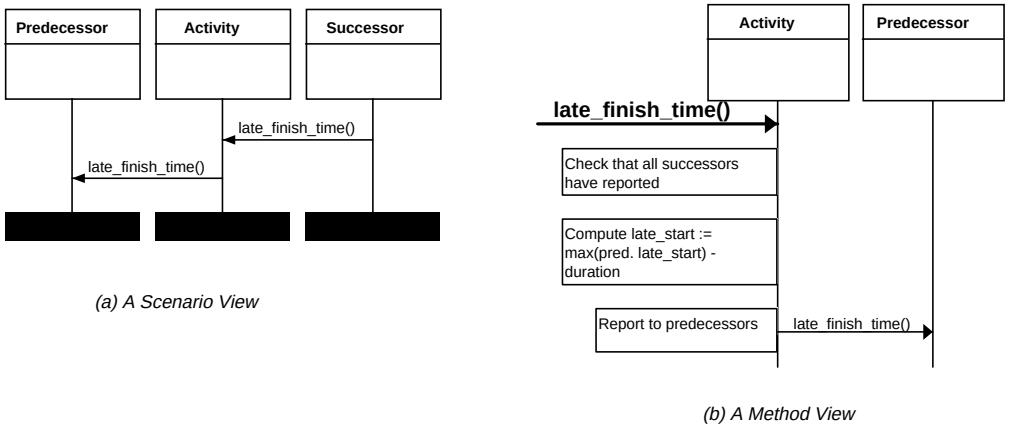

Figure 9. Simple back-loading loading computation

1.1.2.4 Interaction versus service

Figure 10 illustrates that a role model identifies both the sender and the receiver of a message interaction. In a class model, the focus is on message services and the message interface of each class is identified.

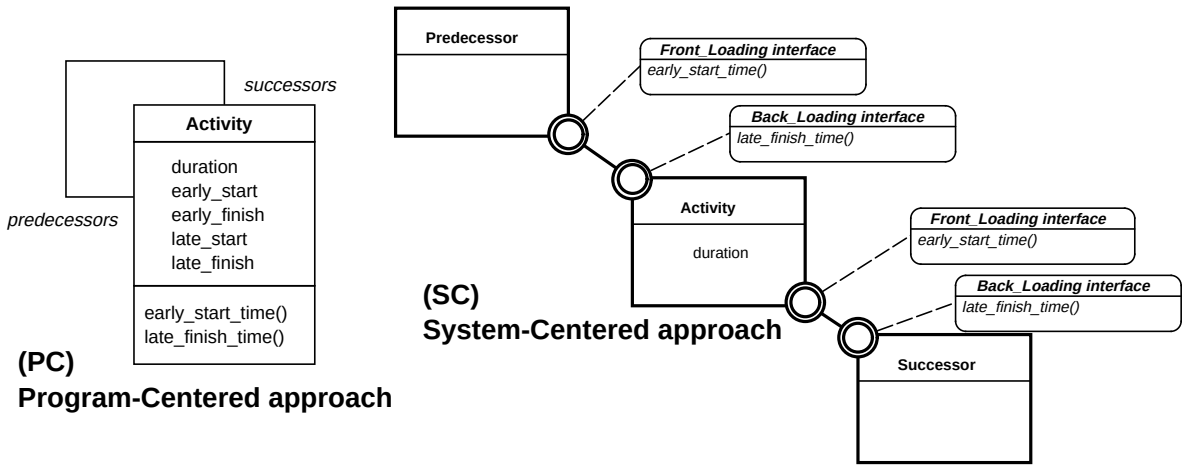

Figure 10. Vanilla network interfaces

The fundamental dichotomy in the description of complex object systems as illustrated in figure 11.

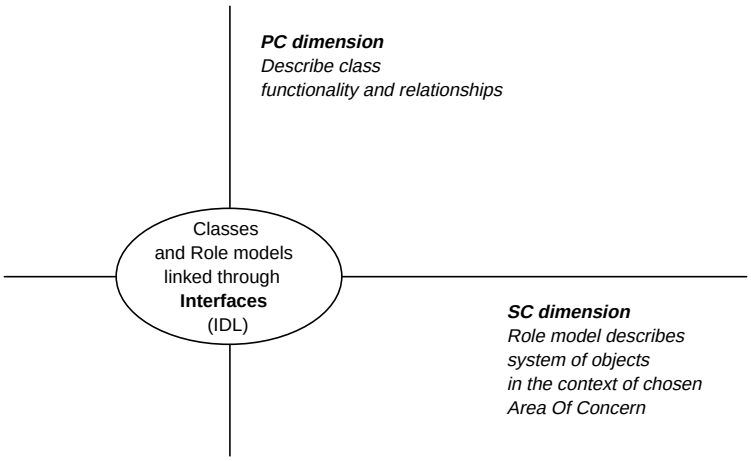

Figure 11. Two dimensions in object modeling

The SC dimension is linked to the PC dimension through interface descriptions. Role model activities consist of objects invoking operations in other objects. These operations are described in interfaces associated with the different interaction paths. Knowing all the roles that are to be played by an object, the interfaces of its class can be created as the union of all the relevant role model interaction interfaces.

1.1.2.5 Class inheritance versus system inheritance

We now introduce the idea of *system inheritance*. With system inheritance, we not only inherit the properties of individual objects and classes, but also the overall system structure and behavior properties.

As an illustration, we will create a specialized cluster of network activities. Consider a home painting operation. Assume that two of the activities are *ApplyCoat1* and *ApplyCoat2*. Default activity rules say that the second activity can only start after the completion of the first. This would be unnecessarily stringent for large paint jobs. We will create a specialized activity cluster where the two paint activity objects negotiate to determine possible time overlaps.

Figure 12 shows how the specialized model is composed by reusing the simple base model twice in two *synthesis* operations. Figure 13 shows how the corresponding frontloading scenario is derived from the base model scenario with an added message for retrieving overlap parameters. Figure 14 shows a specialized method that implements the new algorithm for computing the *early_start_time* for the second coat of paint.

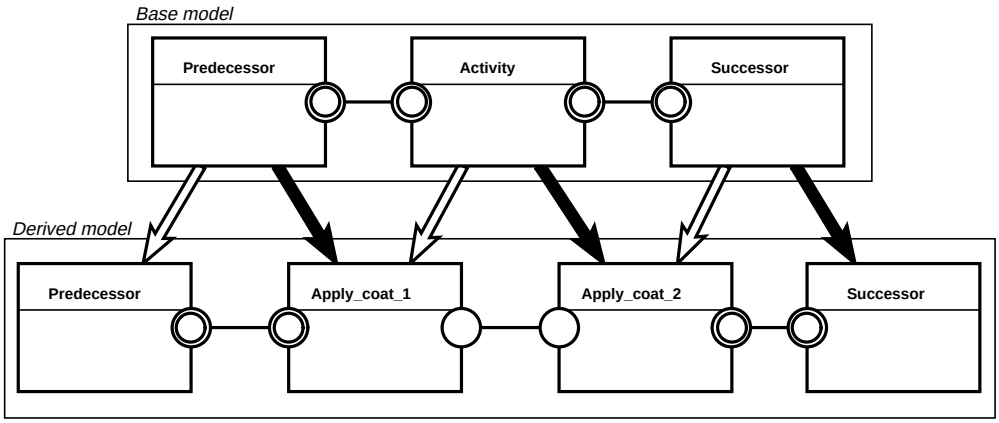

Figure 12. Composite paint activity with overlap calculation

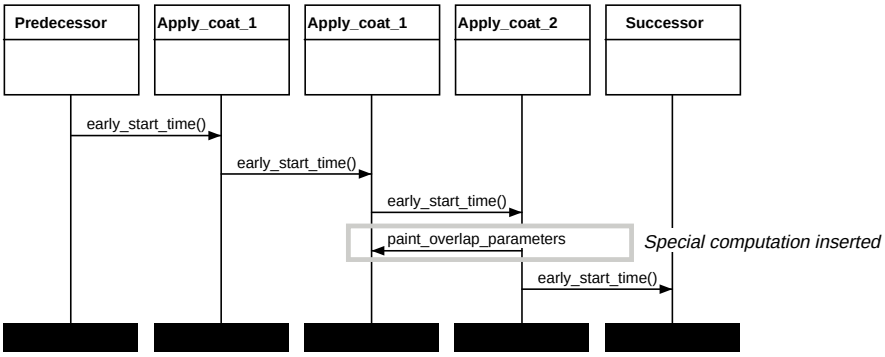

Figure 13. Derived model frontloading scenario

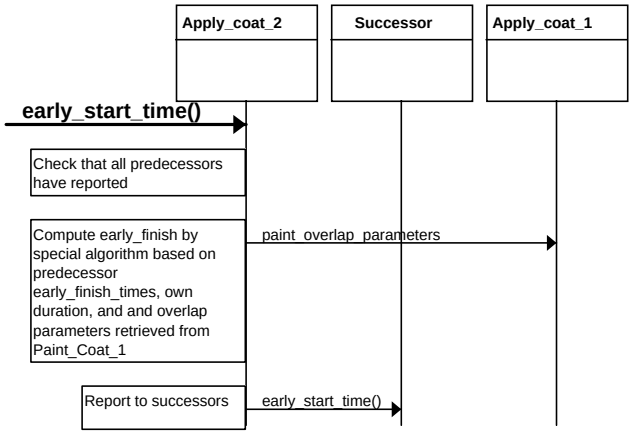

Figure 14. Specialized method

All the static and dynamic properties of a *base role model* are inherited into a *derived role model*. In this *synthesis operation*, all the roles of the base model are mapped onto corresponding roles in the derived model. We thus inherit not only the properties of individual roles, but also their collaboration properties. In *activity superposition*, the Use Cases of the base model are declared to be Use Cases of the derived model. In *activity aggregation*, the Use cases of the base model expand and detail certain operations in the derived model activities.

We see that a role model based reusable component reflects all the benefits of class inheritance. In addition, the role model captures the synergy effect of collaborating objects by capturing their interactions and other dependencies. This is very important in reusable *frameworks*, because we need to capture object dependencies and interactions in a way that preserves the functionality and integrity of the reusable components in the derived products.

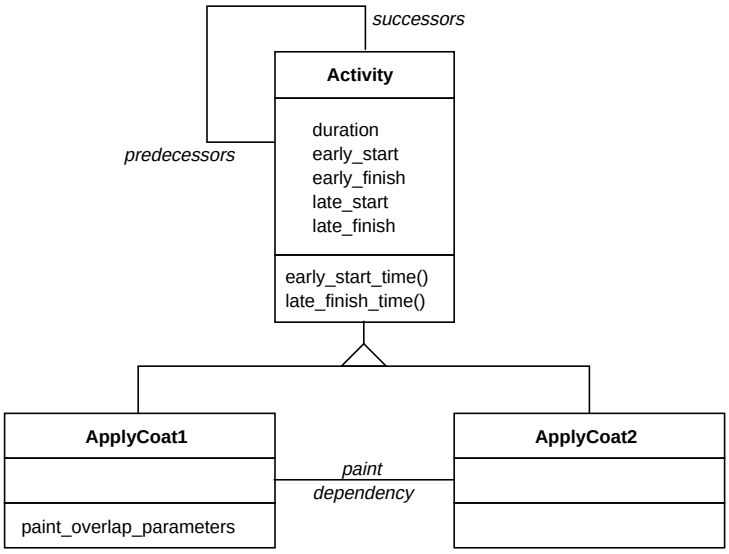

Figure 15. PC view of paint cluster

Figure 15 illustrates the PC approach for describing two coats of paint. Two subclasses are needed. Class *ApplyCoat1* implements the new method *paint_overlap_parameters*. Class *ApplyCoat2* will have a method that calls this new message.

1.1.3 Flexible model requirements

The OOram meta-model is designed for flexibility. The general rule is that any model element that makes sense in a given context shall also be permitted in that context. Conversely, any model element that is not essential in a given context can also be omitted.

Figure 16 gives a rough overview of the dependencies. The detailed dependencies can be derived from the meta-model itself.

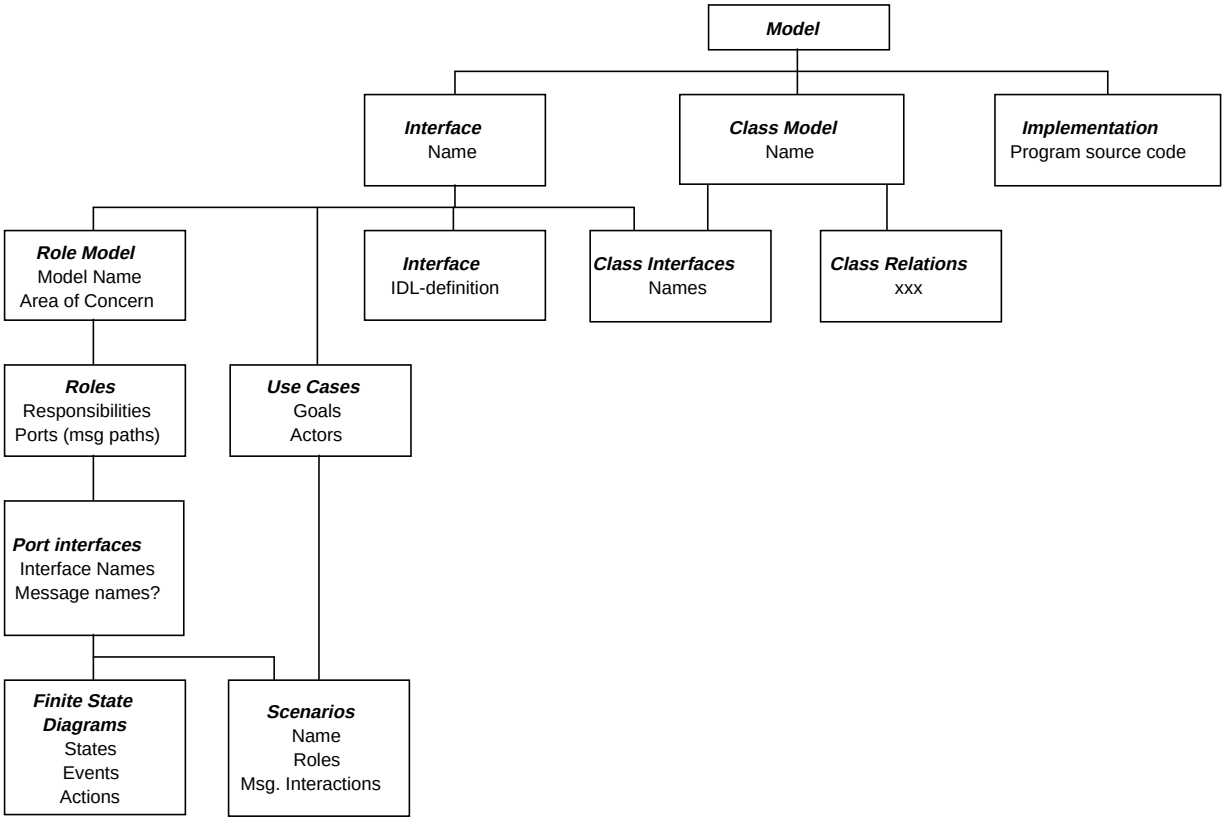

Figure 16. Dependency hierarchy of model elements.

Note: Implementation source code can be written in a programming language without prior formal modeling.

1.1.4 Relationship to other proposals

This section must necessarily be tentative. An obvious reason is that at the time of writing, we had not seen the other proposals in their final form. Equally important: we cannot hope to understand the stated and implicit properties of the other proposals as well as their authors. This section should therefore be read as an invitation to constructive discussions, and not as a comparison between competing solutions.

Most modeling methods distinguish between class models and object models. Object models are particularly useful for describing system behavior, which is not easily expressed in the class domain where objects have lost their identity.

An object has identity, there never has been and never will be another object with the same identity. So an object model element representing an object is very concrete, it represents one particular object. Practical object modeling violates this strict definition with great success; object models are usually made generic by letting the model elements represent archetypical objects in archetypical object structures. Using object models this way yields important benefits for describing a system's dynamic and functional properties.

Object Type is not suitable as the basic building block in a system model. A type is usually defined by a predicate, e.g., a set of operation signatures. All objects satisfying this predicate are instances of the type. Specifying a *use* relationship between two types specifies that some instance of the first type uses some instance of the second type.

The value of a *system* is greater than the sum of the values of its parts. The added value stems from its *structure*, the position of a part relative to the other parts is of prime importance. The parts may be of the same or different types, their function and responsibility in the overall scheme is dependent on their position. A system model, therefore, needs unambiguous representation of object identity. By definition, type and class models cannot fulfill this requirement.

A role model is an abstraction on a structure of collaborating objects. In an instance of a role model, each role represents exactly one object. So the role retains the object identity property without being locked onto a concrete object. Role models formalize what object modeling practitioners have always been doing intuitively. The concept of a role is partly captured in the object model, but the higher conceptual precision is needed to achieve full leverage. By definition, role models satisfy the identity requirement of system modeling.

The most important advantages can be summarized as:

1. *Separation of concern*. Different role models can describe different areas of concern by focusing on different systems of objects. This holds true even when the sub-systems have objects in common. This allows a system to be described in terms of its relevant components, where each sub-problem can be focused by one or more role models. Synthesis permits role models describing different concerns to be composed to provide a more complete view of a system.
2. *Reuse of system models*. A system of collaborating roles can, if carefully designed, provide a pattern that is useful in a variety of situations and systems. This role model can then be reused as a pattern and be specialized for use particular solutions.

Object models retain the object identity property. They can therefore be used as system models. Figure 17 shows how the activity network of figure 5 can be represented by a structure of objects.

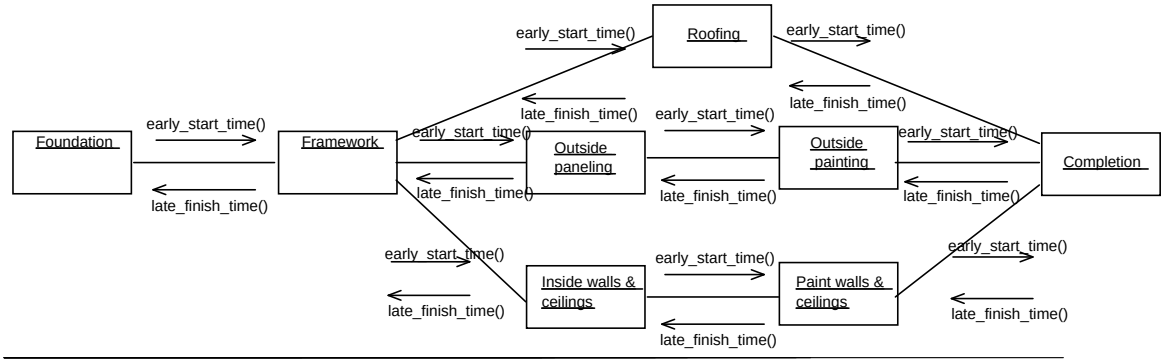

Figure 17. Concrete object representation of activity network

The fundamental concept of an object implies that an object model must describe a system in terms of all objects within the running system. This is of course not desirable, since this implies immensely large models even for modest systems. Instead of modeling concrete object systems, object modeling is focusing on the modeling of archetypical patterns of objects representing a part of the system. In the case of the activity system, the model can for example be described in terms of the generic object model of figure 18

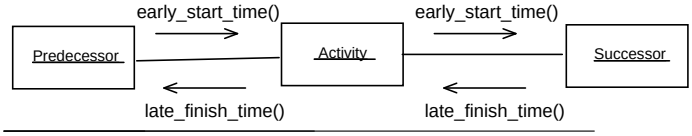

Figure 18. Activity object collaboration (UML)

The UML object model focuses on objects as class instances. It describes typical collaboration patterns of objects, so called archetypical object collaborations. The object model can specify directed messages that can flow between collaborators in the model. In some ways, the supported set of semantics is larger than that of role models. On the other hand, it is without the power of separation of concern and synthesis.

OML [OML] has a quite different approach to modeling than UML, being driven from a responsibility point of view. The OML object model focuses on collaboration between objects and the relationship between objects, classes and roles as illustrated in figure 19. OML has incorporated the concept of role into their framework. A role is a unit with some responsibility, and can be related to either objects, classes, or use cases. The use of roles in OML enables the definitions of patterns that are instantiated by objects. This framework is closer to the role abstraction framework. The relationship between a collaboration of objects and a role pattern can be viewed as a single level of synthesis. Enhancement of this framework with full semantics of the role abstraction (synthesis of role models) should yield a more powerful modeling framework.

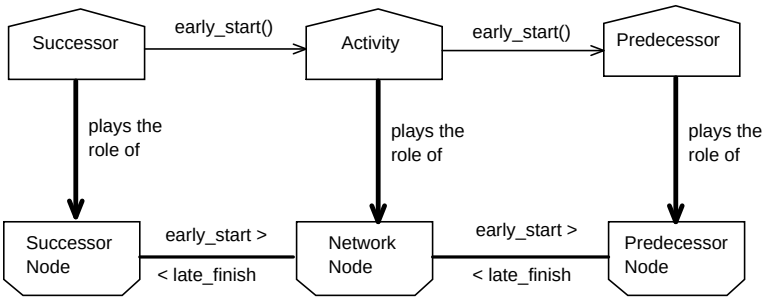

Figure 19. An OML object collaboration shows objects in interaction and a corresponding role pattern.

The object collaboration model can be described in a corresponding role model. The main difference is that the role model has explicit ports to which interfaces are assigned. There is a relatively small, but significant, conceptual gap between the semantics of role models and the current, informal semantics of object models. These differences relate to the treatment of the system model as a first class citizen, the treatment of object identity, and the inheritance of whole system models.

Figure 20. Activity role model

The role model interfaces are precisely defined in IDL:

```
interface FrontLoading {
  void early_start_time(in integer receiver_earliest_start_time);
  };

interface BackLoading {
  void late_finish_time(in integer receiver_latest_finish_time);
  };
```

We see that the role model not only shows the messages that must be understood by an object, but also identifies their senders and the paths used for object interaction in the system's processes.

The models described above represent three different levels of abstraction

1. Concrete objects in figure 17.
2. Object model with informal roles in figures 18 and 19.
3. Role model in figure 20.

When looking at the two last model abstractions we notice that they have a similar way of modeling the activity problem. The activity network is, however, only one instance of a more generic problem; the network analysis problem. Another instance of this problem could be the travel network problem illustrated in figure 21.

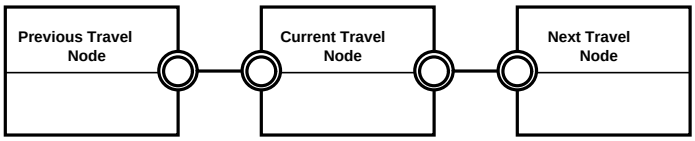

Figure 21. Travel network role model

Two similar problems have been shown here; they are described by almost identical role models. The problem illustrated by the activity and travel networks has a general abstraction, a generic network model, consisting of network nodes with predecessors and successors (nodes) with a well-defined behavior. This abstraction can be modeled in terms of a generic role model, a network model pattern which describes the general network abstraction. The activity and travel specialization models can be defined by synthesis of the generic solution, as a pattern of roles.

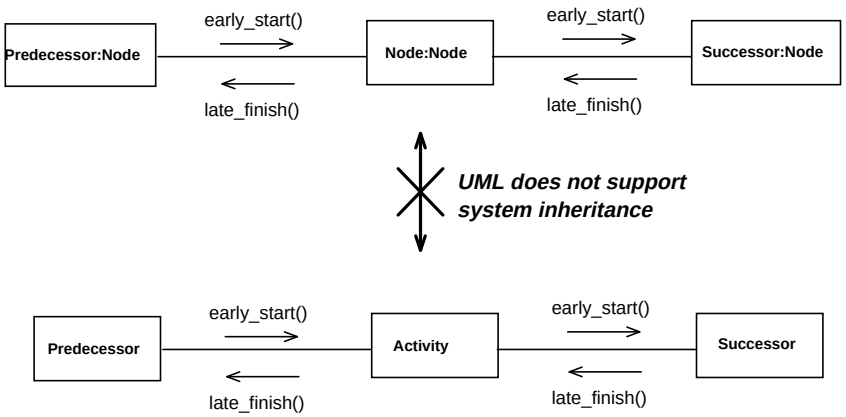

Figure 22. UML object interaction model for general network.

As figure 22 illustrates, UML can describe the network models in terms of separate object collaborations. It does not, however, enable the abstraction and connection between models in terms of synthesis that is illustrated for role models in figure 23. A generic pattern model is maintained and possibly updated as a separate model. The specialization only specifies the properties added or renamed.

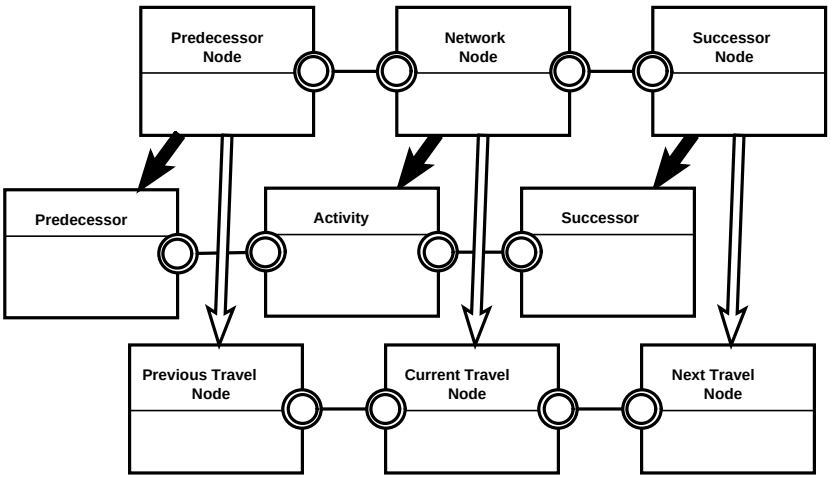

Figure 23. Role flexibility and power by synthesis

The concepts involved in role modeling and the abstractions it provides are at different degree partly incorporated in the UML and OML methodologies. The semantics of role modeling can, however, be integrated into these methodologies. Performing this integration would imply an extension of the current metamodels of these methodologies.

For UML, the extensions would imply:

1. Incorporating the concept of role models into the metamodel as an abstraction of object collaborations
2. Incorporating the concept of role into the metamodel as an abstraction over objects
3. Incorporating the notion of synthesis of role models into the metamodel by adding inheritance of role (object) collaborations.

For OML, the extensions would imply:

1. Realizing the full capabilities of the role and role model abstractions as something more than a placeholder for responsibilities.
2. Incorporate the full notion of role model synthesis, e.g., by specializing the OML cluster inheritance concept. By viewing a "role-cluster" as a collaboration of roles for some concern, clusters of roles can be "synthesized" by allowing multiple inheritance of "role-clusters"

The Catalysis methodology [Cat] is founded on four central concepts: types, conformance, collaborations, and frameworks. Types focus on object behavior in terms of interfaces, conformance is a foundation for traceability, collaborations are sets of actions between typed objects, and frameworks define generic patterns by using placeholders.

Catalysis engages a number of concepts similar to our proposed meta-model as well as other powerful formalized analysis and design concepts such as formal specifications and constraints. Our proposal would be enriched by the inclusion of such concepts. The collaboration model in Catalysis describes actions between typed objects. The types in collaboration represent roles played by objects. We feel this confusion of role and type may be unfortunate, see the argument about object identity above.

Another difference between Catalysis collaboration concepts and our proposed role-models is caused by Catalysis' strict usage of types to represent roles. The type exposes its full interface to all its collaborators. It describes role responsibilities in terms of the type of the role (its complete interface) and not in terms of what a role is allowed to do on its different communication paths. Catalysis frameworks are generic abstractions of type relationships (and collaborations) where placeholders are used to provide generic representations of types. These frameworks are reusable assets which focus more on types, relationships, and implementation classes than roles with identity. The semantics of synthesis is not captured by the Catalysis frameworks because of the different level of abstraction.

IBM and ObjecTime have jointly prepared a proposal for the OMG OA&D RFP-1. This proposal describes a generic metamodel aimed at capturing the semantics of all object models. To support formal specification of the objects in the model, the formal Object Constraint Language (OCL) is used for constraints and queries. This language would also be useful for describing static constrains in all object models, including application models and our meta-model. Other abstractions seem to be needed to capture the dynamic properties of systems and system inheritance. Egil Andersen of the University of Oslo is working on such abstractions based on labeled transition systems (LTS). Unfortunately, this work is not yet completed so it cannot be submitted to OMG at this time.

The role-model concept is not explicit part of the IBM/ObjecTime proposal, but can be incorporated by way of patterns and refinement of patterns within the metamodel. It is believed that the idea of supporting only the universally common set of object semantics can include roles and role-models, because the semantics of these can be found within every system of objects. We are not convinced. We believe it to be insufficient to specify the objects, and that some abstraction on *the system as a whole* should be included as a first class citizen. In addition, we believe that leverage implies some form of rigidity. If the standardized meta-model is too general, important benefits can be lost. The emerging OMG standard should be sufficiently precise to provide optimal leverage for other OMG activities such as the efforts for standardizing Business Objects and various Domains.

In summary, the strict object level is not a feasible modeling level. The intuitive, generic object model abstraction level has been chosen as an abstraction in many methodologies. There are, however, important advantages with addressing the role abstraction to its full extent, notably the separation of concern and the use of synthesis to achieve system inheritance.

1.2 Proof of Concept

RFP-1 Requirements:

1. Submissions must include a "proof of concept" statement, explaining how the submitted specifications have demonstrated to be technically viable. (p.13)

Reich Technologies is an international group of companies, providing a coordinated suite of products and services to support Object-Oriented software development in large corporations. With a presence throughout Europe and North America, Reich Technologies occupies leading positions in the world markets for integrated OO CASE-Tools, fine-grained object repositories and OO team programming environments.

The Intelligent Software Factory (ISF) offers an integrated Object-Oriented CASE-Tool suite. It is built on the concept of Model-Driven Development in which the work done at the beginning of a project creates an environment for configuration management and cost containment for software maintenance. The Intelligent Artifact Repository (IAR) provides an enterprise-wide resource for the management and reuse of Information System assets. This concept is so powerful that the development team uses ISF and IAR for production, making ISF the first CASE tool to be self-generated.

Recognizing the impact of introducing tools, Reich Technologies offers success oriented services including training, consulting and tool customizations. Corporations combine tools, services and processes with their own organizations to implement a Corporate Software Ecology.

Reich Technologies worked with Alistair Cockburn and Ralph Hodgson to flesh out the concept of Use Case and integrate it in the context of Responsibility-Driven Design. Several large companies have built systems upon these constructs since 1992. Structured UseCases and detailed Responsibility models proved to be a relevant answer to the challenge of gathering and organizing thousands of requirements, defining the scope of the system, and designing an architecture for objects. A methodology with processes and identified deliverables has been created in a joint effort.

As tool builders, Reich Technologies adds the knowledge of implementing lifecycle management for the meta-model objects. Reich Technologies has also extensive experience designing the meta-models that implement in ISF the modeling notations of diverse methodologists.

Reich Technologies sells off-the-self and tailored versions of their CASE-Tools. It is a profitable, privately-owned company.

Humans and Technology, Inc., is a privately owned design, consulting and teaching company founded in 1994 by Alistair Cockburn. It specializes in disseminating object-oriented design and project management, primarily through attention to human social, psychological and cognitive factors. Alistair Cockburn was author of the IBM Consulting Group's OO software development methodology produced in 1992 and revised in 1994.

The techniques and recommendations are based on project debriefings taken since 1991, and tested on live projects, within and outside IBM, and in Hong Kong, New Zealand, the U.S., Canada, and Europe.

Use cases, interactions, and responsibilities have been constant cornerstones of the recommendations since 1992.

Use cases, as described, were applied to a 2-year, 40-developer project having over 200 such use cases. They are supported in the described form in the Intelligent Software Factory tool from Reich Technology, and in similar forms in Software Through Pictures and Rational Rose. They have been described by Ivar Jacobson, and in more detail by Alistair Cockburn. They are taught by most design, consulting and teaching firms. The cited model is in use in IBM, Lucent / AT&T, Hewlett-Packard, Ralston-Purina, OOCL, and other companies.

Responsibilities and interaction have been in use since the late 1980's. In project debriefings they emerge consistently as a highly rated way to briefly described a system consisting of classes, objects or subsystems. They are directly supported in the Intelligent Software Factory and directly or indirectly supported in other tools. They are described in several books.

Taskon A/S is a privately owned Norwegian company founded in 1987. It is based on a series of object-oriented research and development projects at the *Center for Industrial Research* in Olso, Norway from the mid-1970s to the end of the 1980s. (The Center for Industrial Research is now part of the SINTEF group).

The current OOram method and tools are based on these research results, on the experiences gained by Taskon and its customers, on the results of research at the Universities of Oslo and Trondheim, and on Taskon's own research and development.

The OOram role modeling concepts and tools have evolved successfully from 25 years of experience with object oriented system development within areas such as enterprise modeling, information management systems, real time systems, general software development, etc. The OOram method and tools are heavily oriented towards managing complex systems of collaborating objects; providing powerful leverage for separation of concern and for the synthesis of larger systems from simpler components for reusable frameworks and other software assets. development.

The OOram method is thoroughly described in [Ree 96].

OOram Professional 4.0, a tool supporting the OOram method, is currently used by customers such as ABB, Alcatel, Deutsche Telekom and IBM. An evaluation copy of the tool is available from Taskon's web page located at: *http://www.sn.no./taskon.*

The method and tool development has been financed by Taskon, public funding, EU projects, and Taskon customer projects. Taskon is a profitable, privately owned company that was established in 1987.

The following European collaborative R&D projects in the Esprit programme are using OOram and role modeling:

1. *DISGIS - Distributed Geographical Information Systems* - focusing on ISO ODP and CORBA + COM/OLE for Interoperable GIS, using OOram role modeling and ISO/TC211 and OGIS standards - with Iona, Taskon, GIS Denmark, Norwegian Mapping Agency, Sysdeco DIKAS, INESC and SINTEF.
2. *ECHO - ECDIS Chart Hub Operation* - focusing on digital updates to Sea Map Databases on ship via Satellite link to Chart centers, using CORBA and role model bases analysis and design, with Iona, Norwegian Hydrographic Office, Sysdeco DIKAS, Finnish Maritime Agency, British Admirality and SINTEF.
3. *OBOE - Open Business Object Environment* - focusing on implementing and applying the OMG Business Object Facility for Business Objects in the domain of POSC Oil&Gas Exploration&Production - Seismic Processing - creating a role model based business object modeling methodology - with SSA OT, Prism, Schlumberger-Geco, Open Group, University of Frankfurt, Heinz Nixdorf Institute and SINTEF.
4. *OMEGA* - focusing on CORBA based architecture for a geometry/topology package using OMG and POSC EpiCentre standards, combined IDL and EXPRESS and OOAD modeling in the Oil&Gas Exploration&Production domain - Reservoir modeling - with Matra Datavision, TNO, BEICIP, BRGM, Volumetrix and SINTEF.
5. *SEASPRITE - Software Architectures for Ship Product Data Integration and Exchange* - focusing on CORBA-based and STEP/EXPRESS based information sharing and exchange for Ship Design and Building - with Kværner, Odense, Vulkan, Lloyd, Det Norske Veritas, Kockums, MARIN and SINTEF.
6. *SUPREME* - focusing on an integration framework, process model visualisation and OOram role modeling for a WfMC based Workflow system with CORBA interface to existing applications - with Cap Gemini Innovation, Taskon, University of Linkoping, Swedish Energy and SINTEF.
7. *COMPASS - Component-based Accounting System and Services (a proposal)* - focusing on modeling (with OMG OOAD/role models) and creating and applying an Accounting/General Ledger Facility - with Stanford Software, Economica, Visma, Software Box, Real Objects, LogOn and SINTEF.

1.3 Organization of Proposal

The concepts of the meta-model are discussed in chapter 2. The corresponding notation is discussed in chapter 9. Resolution of technical and non-technical issues is done in chapter 3. Specification Dependencies, Relationship to CORBA, Relationship to OMG Object Model, Standards conformance, and Other Information are discussed in the following chapters.

More specifically, some RFP-1 Requirements are discussed as follows:

1. *Submission must identify where meta-model specification (and possibly notation) are discussed.* See chapter 2.
2. *Submission must identify where meta-model extensibility is discussed.* See chapter 1.1.
3. *Submissions must define the interfaces that enable models or model elements created by one tool to be accessed from another tool. (p.19).* Meta-model interfaces are specified in IDL in chapter 2.5.
4. *It should be possible to separately specify and implement each specification (p.11).* There is only one specification of a comprehensive and consistent meta-model. Its parts can be included or excluded as needed. See also chapter 1.1.3.
5. *OMG specifications shall be complete (p.11).* Chapter 2 includes all the specifications needed to navigate among the objects of a model and to retrieve object attributes.
6. *Minimize duplication of functionality (p.11).* Required functionality is referenced, not duplicated.
7. *OMG specifications shall not contain implementation descriptions (p.11).* The meta-model does not in any way bind its implementation except that its objects shall be accessible through CORBA.
8. *No hidden interfaces among specifications (p.12).* All required interfaces are given explicitly in chapter 2.5
9. *Shall use OMG IDL naming conventions for interfaces, types, operations, attributes, etc. (p11).* The proposal uses OMG IDL naming conventions for interfaces, types, operations, attributes, etc.
10. *Use multiple inheritance and multiple interface capability of OMG IDL whenever possible. (p.12).* Multiple inheritance has been used in the few cases where it was necessary, see chapter 2.5.
11. *OMG specifications should have precise descriptions of semantics and side-effects. (p.11).* Precise description, though not formal descriptions, have been included. The meta-model describes read-only access to an OA&D model; it is therefore safe of side-effects.
12. *Define, explain, and/or demonstrate mandatory versus optional interfaces. (p.13).* See chapter 1.1.3.

13. *Operation sequencing shall be included where applicable (p.11).* There are very few sequencing restrictions since the model is a read-only model. The only exception used is the standard OBJECT_NOT_EXIST.

1.4 The Submitting Companies

Taskon A.S. is a Norwegian company formed in 1988 for the purpose of supporting various categories of end users with individually tailored information systems by utilizing object-oriented technology. The company builds on technology developed during the past 15 years at SINTEF and at Taskon with an investment of approximately 250 man-years. Taskon is a supplier of information, products and consultants for object-oriented system development.

For more information about Taskon's products please contact:

Taskon AS
Gaustadalleen 21
N-0371 OSLO
NORWAY
Telephone: +47 22 95 86 31
Fax: +47 22 60 44 27
Email: info@taskon.no
http://www.sn.no/taskon/

Reich Technologies is an international group of companies, providing a coordinated suite of products and services to support Object-Oriented software development in large corporations. With a presence throughout Europe and North America, Reich Technologies occupies leading positions in the world markets for integrated OO CASE-Tools, fine-grained object repositories and OO team programming environments.

The Intelligent Software Factory (ISF) offers an integrated Object-Oriented CASE-Tool suite. It is built on the concept of Model-Driven Development in which the work done at the beginning of a project creates an environment for configuration management and cost containment for software maintenance. The Intelligent Artifact Repository (IAR) provides an enterprise-wide resource for the management and reuse of Information System assets. This concept is so powerful that the development team uses ISF and IAR for production, making ISF the first CASE tool to be self-generated.

Recognizing the impact of introducing tools, Reich Technologies offers success oriented services including training, consulting and tool customizations. Corporations combine tools, services and processes with their own organizations to implement a Corporate Software

Ecology.

More information about Reich Technologies and OMG OAD proposal is available on the Internet http://www.projexion.com.

Reich Technologies US
Cecilia Schuster, VP of Operations
PTEC 209, 19E Central Ave.
Paoli, PA, 19301, USA
+1.610.889.9606
fax +1.610.640.0619
Æ100437.3555@compuserve.comÅ

Reich Technologies Europe
Georges-Pierre Reich, Chief Architect
Æ100431.124@compuserve.comÅ
Agnes Guichard, International Public Relations
Æ106003.2514@compuserve.comÅ
40 Quai de la Douane
29200 Brest - France
+33.2.98.80.48.99
fax +33.2.98.44.77.72

Humans and Technology, Inc., is a privately owned design, consulting and teaching company founded in 1994. It specializes in disseminating object-oriented design and project management, primarily to larger OO projects.

For more information about Humans and Technology, please contact:

Humans and Technology
attn: Alistair Cockburn, President
7691 Dell Rd
Salt Lake City, UT 84121
USA
+1.801.943-8484
(fax: +1.801.943-8499)
(email: arc@acm.org)
(http://members.aol.com/acockburn)

Chapter 2
Conceptual Meta-Model

The meta-model proposed in this document is basically described in terms of its own concepts and notation. The concepts are defined in this chapter. The corresponding notation is defined in chapter 9: *Model notation.*

We define the following terms:

> **model**: A simplified representation of a real-world concept used to define an application, a tool, a business, or a technology.
>
> **meta-model**: A model of the modeling concepts used to define models.

A model is considered as a coherent and consistent system of objects. These objects are basically organized in a tree structure. Additional cross references augment the model semantics and coherence. Objects in the system environment navigate in the system object structure in order to access the system objects.

All model elements have the common attributes *element_name, element_explanation,* and *named_characteristics:*

```
struct DictionaryData {
  string keyword;
  string value;
  };

typedef sequence<DictionaryData> Dictionary;

interface ROElement {
  string element_name();
  string element_explanation();
  Dictionary named_characteristics();
  };
```

This proposal defines a read-only meta-model. If it were to be extended with editing capabilities, each parent object in the tree would be responsible for managing the life cycle of its child objects; i.e., their creation, storage, and ultimate destruction. Each child object would be responsible for managing cross references and references from the environment. For completeness, this basic management mechanism is indicated in Appendix 1: *Element life cycle management subsystem.*

The root of a model tree represents the model as a whole. This object can be managed by the Object Request Broker [CORBA], or it can be retrieved from any application object that wishes to divulge its logical structure. Section 2.1: *Meta-Model Architecture* specifies how a client of an application object can retrieve reference to a model object.

We propose three major abstractions: The *interface*, the *role model*, and the *class*. These abstractions are complimentary and mutually consistent. Each abstraction is represented as a subsystem that highlights certain model properties and hides others. The semantic relationship between the abstractions is illustrated in figure 24(a), while figure 24(b) shows that an object representing a role can find the relevant interfaces and implementing classes. Similarly, an object representing a class can find its implemented interfaces and specifying roles. Finally, an object representing an interface can find the roles where it is referenced and the classes where it is implemented.

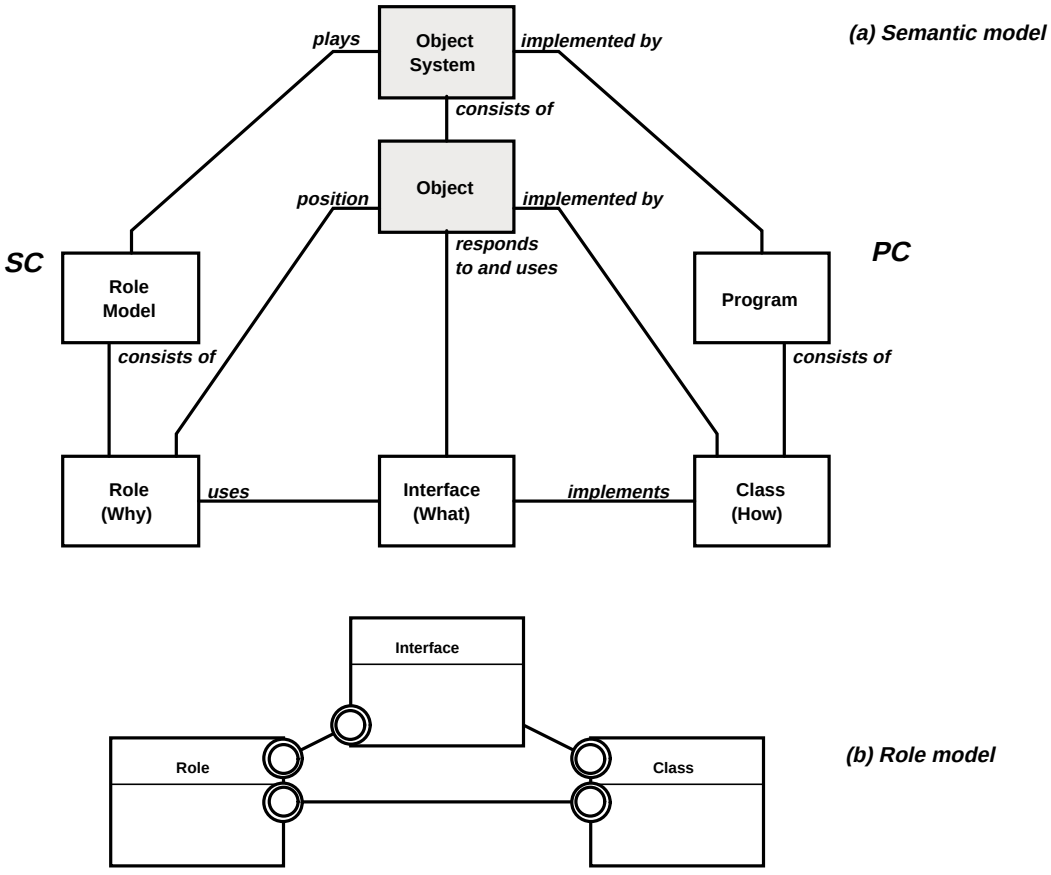

Figure 24. Semantic and referential relationships between the object and its major abstractions

In the following subsections, we will describe the different abstractions as a set of subsystems. For each, we will show its structure and specify the names of the interfaces on each of the interaction paths.

The subsystems are as follows:

1. The *Meta-Model Architecture* describes the root of the model tree with its client and its immediate substructure. (Section 2.1)
2. The *Interface meta-model* describes the model object structure and operations representing an OMG Interface. (Section 2.2)
3. The *System-Centered meta-model* describes the model object structure and operations representing a role model. (Section 2.3)
4. The *Program-Centered meta-model* describes the model object structure and operations representing a class. (Section 2.4)
5. The *Element life cycle management subsystem* describes the object structure and operations that manages object creation, operation, and destruction. The Object life cycle management subsystem model is a base model, the above five models could be derived from it if they were to be extended to read-write models. (Appendix 1)

The model collaboration structure is described in a number of role models. The capabilities of the model objects as seen from the read-only clients are specified in the corresponding interfaces. The interface definition is given in section 2.5.

2.1 Meta-Model Architecture

2.1.1 Rationale

The main purpose of the OA&D facility as specified in RFP-1 is to *"play a key role in achieving semantic interoperability between OA&D tools"*. In this proposal, we have made a number of additional assumptions:

1. System models are to be represented as systems of objects. The OMG is about the application of objects, we see no reason why the system models should be an exception. The system model is, therefore, a system objects.
2. The model of a system model, called the *meta-model,* is expressed in the same manner as all other models. The concepts and notation used for the description of the meta-model are the concepts and notation defined by the meta-model. The meta-model is an instance of itself.
3. Models are observable through the ORB, making them generally available to different kinds of clients.
4. The clients of system models are described in the RFP-1 as being various OA&D tools. We believe there are other, potentially more valuable, uses of a meta-model supported by a model service facility:
   - We add the clients of application systems as an important extension. By giving a client access to the application system model, this client can adapt to the specific characteristics of that model. It will also be feasible to provide automatic generation of at least some parts of the client programs.
   - OMG and other relevant industry bodies can publish recommended object structures in the form of system models that can be specialized and implemented.
   - Software vendors can offer ensembles of classes that implement the recommended structures so that application developers can derive their solutions from these object frameworks.
   - Application developers derive their designs by specializing the recommended object structures, and possibly also derive their implementation classes from the available framework classes. They then create their application and publish its model so that clients can adapt to it.
5. The meta-model supports a complete set of abstractions without mandating any of them. This permits the modeler to choose the particular set of abstractions that best communicate the ideas that are important in the given context.

Application clients need read-only access to the Application model, this is illustrated in figure 25 (a). In (b), two *OA&D tools* marked *Tool-A* and *Tool-B* share a common model repository. This puts very strong demands on the *Application model*, since it must support all editing operations for both tools.

In alternative (c), each tool has its own model. *Tool-B* reads model data from *Application model-A,* and writes these data into its own *Application model- B* where it can edit them. Similarly, *Tool-A* can read model data from *Application model-B,* and write these data into its own *Application model-A* where it can edit them. As in case (a), we only need standardize model structure and operations for reading information from the model. The model edit operations can be specialized for each tool, permitting toolmakers to optimize their products for different development processes.

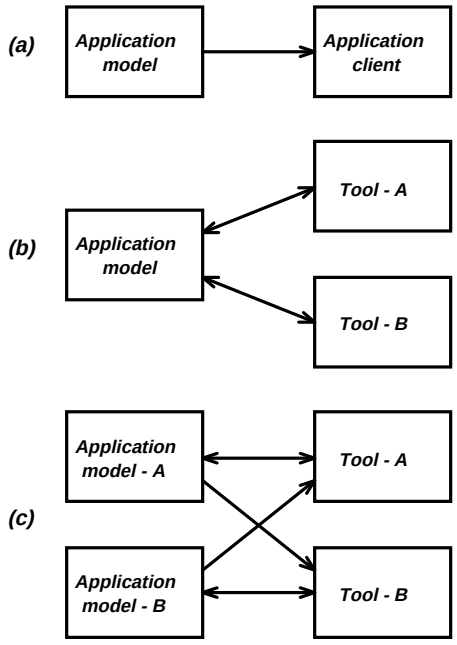

Figure 25. Model access requirements

The meta-model presented in this proposal focuses on model structure and read operations, thus giving priority to alternatives (a) and (c). This is reflected in the chosen interface names, they all being with RO (Read-Only).

2.1.2 Conceptual Model

The system of objects representing a model can be accessed from one of the application objects that it describes, or directly through CORBA. The root of the model is an object playing the *Model* role in figure 26. Its children are the roots of the subsystems representing the main abstractions.

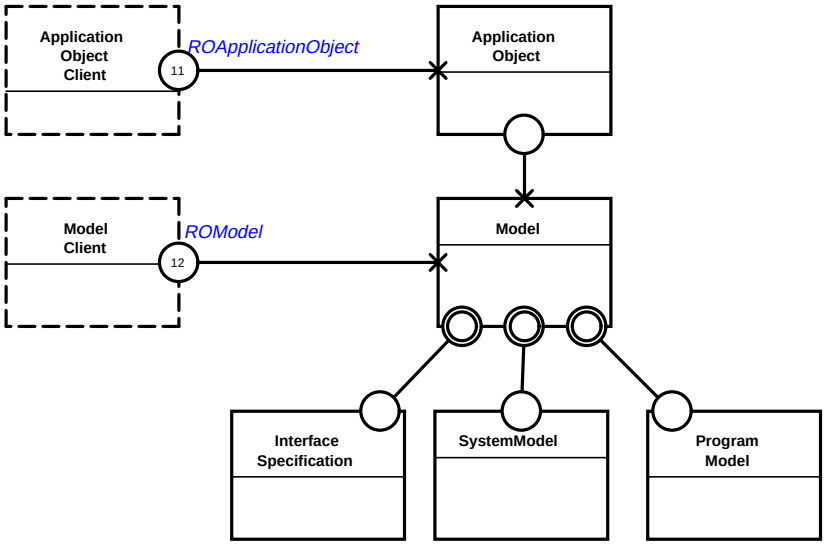

Figure 26. The meta-model collaboration view

The role responsibilities are as follows:

1. *ApplicationObjectClient.* This environment role represents an Application client that wants access to the application model.
2. *ModelClient.* This environment role represents an an object that will start navigating in the model.
3. *ApplicationObject* This role represents an object that manages an application object structure that includes a self-referencing feature.
4. *Model.* An object playing this role represents the model of an application object structure. If this structure is itself a model, the role represents its meta-model.
5. *InterfaceSpecification.* The root of a subsystem describing the model interfaces.
6. *SystemModel.* The root of a subsystem describing the functional aspects (use cases, responsibilities, and role models).
7. *ProgramModel.* The root of a subsystem describing the model classes. Actually, it is meant as a rudimentary object model of UML.

2.1.3 Interface Descriptions

```
interface ROApplicationObject { /* Port # 11 */
  ROModel model(out boolean has_model);
  };

interface ROModel : ROElement { /* Port # 12 */
  sequence<ROInterface> interfaces();
  sequence<ROSystemModel> system_models();
  sequence<ROProgram> program_models();
  };
```

2.2 Interface meta-model

NOTE: The form of the IDL interface language standard is textual. An object model of an IDL interface could, for example, be a parse tree. Here, we have elected to display explicit objects for the concepts needed by the other abstractions. The full interface information is represented as textual definition attributes in the relevant objects.

2.2.1 Rationale

The *Interface* is illustrated in figure 27. [CORBA] defines an interface as follows:

> An **interface** is a listing of operations and attributes that an object provides. This includes the signatures of operations, and the types of the attributes. An interface definition ideally includes the semantics as well. An object *satisfies* an interface if it can be specified as a target object in each potential request described by the interface.
>
> **Interface inheritance** is defined as the construction of an interface by incremental modification of other interfaces. The IDL language provides interface inheritance.

Implicit in this definition is that the receiver object must *at least* provide the specified operations; while the sender object may *at most* invoke this set.

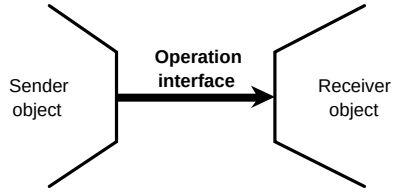

Figure 27. The Interface abstraction

2.2.2 Conceptual Model

The *interface* abstraction is the weakest, and therefore the most flexible abstraction of the proposed meta-model facility. By minimally specifying the interface from one object to another, it leaves open the exact nature of these objects. If a larger scope is needed to describe interesting system properties, interface descriptions may be augmented by role models, and/or class models in a coherent manner.

The role model in figure 28 shows the visible constituents of an interface specification.

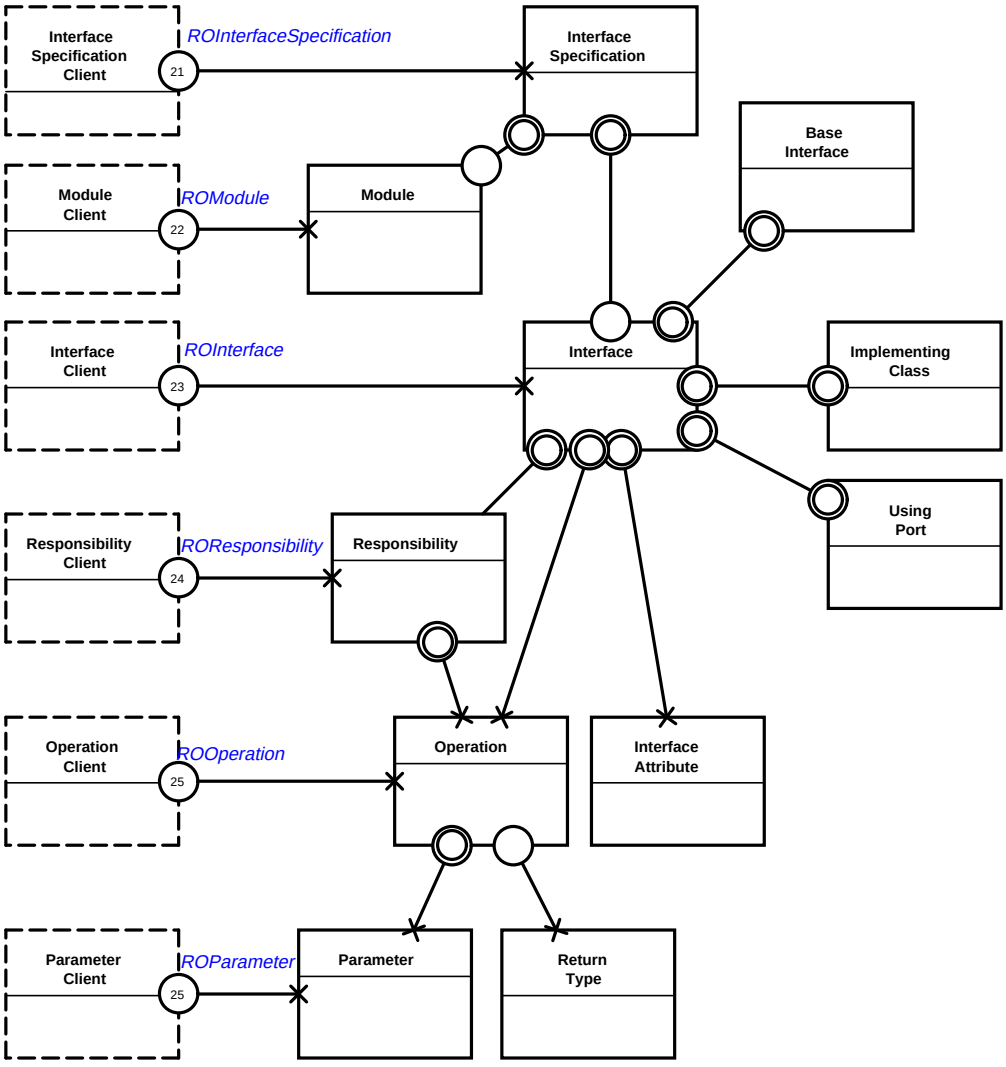

Figure 28. Collaboration view

Role responsibilities

1. *InterfaceSpecificationClient.* Represents a client needing to traverse and examine the IDL specification.
2. *ModuleClient.* Represents an object referencing an object playing the *Module* role.
3. *InterfaceClient.* This role represents an object referencing an *Interface* object.
4. *ResponsibilityClient*. This role represents an object referencing a *Responsibility* object.
5. *OperationClient.* This role represents an object referencing *Operation* object.
6. *ParameterClient.* This role represents an object referencing *Parameter* object.
7. *InterfaceSpecification.* The root of an IDL specification
8. *Module.* Represents an IDL module declaration.
9. *Interface.* Represents an IDL interface declaration.
10. *BaseInterface*. Inheritance relationships may be defined between Interface objects.
11. *ImplementingClass*. Represents classes that implement this interface.
12. *UsingPort*. Represents role model port that uses this interface.
13. *Responsibility*. Represents a responsibility that is supported by this interface.
14. *Operation*. An operation, commonly also called a *message.*
15. *InterfaceAttribute*. An operation, commonly also called a *message.*
16. *Parameter*. A Type in the role of one of the operation's parameter types.
17. *ReturnType*. A Type in the role of the operation return value.

2.2.3 Interface definitions

```
interface ROInterfaceElement : ROElement {
  string definition();
  };

interface ROInterfaceSpecification : ROInterfaceElement { /* Port # 21
*/
  sequence<ROModule> int_modules();
  sequence<ROInterface> int_interfaces();
  sequence<ROType> int_types();
  sequence<ROConst> int_constants();
  sequence<ROException> int_exceptions();
  };

interface ROModule : ROInterfaceElement { /* Port # 22 */
  sequence<ROInterfaceSpecification> scoped_interface_specification();
  };
```



```
interface ROInterface : ROInterfaceElement { /* Port # 23 */
  sequence<ROOperation> int_operations();
  sequence<ROResponsibility> int_responsibilities();
  sequence<ROInterface> int_base_interfaces();
  sequence<ROType> int_types();
  sequence<ROConst> int_constants();
  sequence<ROException> int_exceptions();
  sequence<ROPort> using_ports();
  sequence<ROClass> implementing_classes();
  };

enum InvocationSemantics {Synchronous, Asynchronous,
SynchronousDeferred};

interface ROOperation : ROInterfaceElement { /* Port # 24 */
  boolean op_is_oneway();
  ROType op_return_type();
  sequence<ROParameter> op_parameters();
  InvocationSemantics semantics();
  };

interface ROType : ROInterfaceElement {
  };

interface ROConst : ROInterfaceElement {
  };

interface ROException : ROInterfaceElement {
  };

enum ParameterDirection {direction_in, direction_out,
direction_inout};

interface ROParameter : ROInterfaceElement { /* Port # 25 */
  ParameterDirection parameter_direction();
  ROType parameter_type();
  };
```

2.3 System-Centered meta-model

This meta-model has been split into several submodels using the principle of separation of concern. The dependencies between the models are through objects playing two roles: An object playing a leaf role in one model also plays the root role of another. The model hierarchy is as follows:

1. Meta-model architecture (Section 2.1)
    - System-centered meta-model. (Section 2.3)
        - *Functional subsystem*. Use cases and responsibilities. (Section 2.3.2.1)
        - *Role Model subsystem*. (Section 2.3.2.2)
            - *role model scenario*. A sample interaction sequence. (Section 2.3.2.3)
            - *finite state machine* (*FSM*). There is one for each role. (Section 2.3.2.4)
            - *role model synthesis.* Showing the inheritance relationship between role models. (Section 2.3.2.5)

2.3.1 Rationale

Role modeling is based on an important idea from systems theory: the value of a system is greater than the sum of the values of its parts. The role model describes a system of collaborating objects as a corresponding structure of collaborating roles as indicated in figure 29.

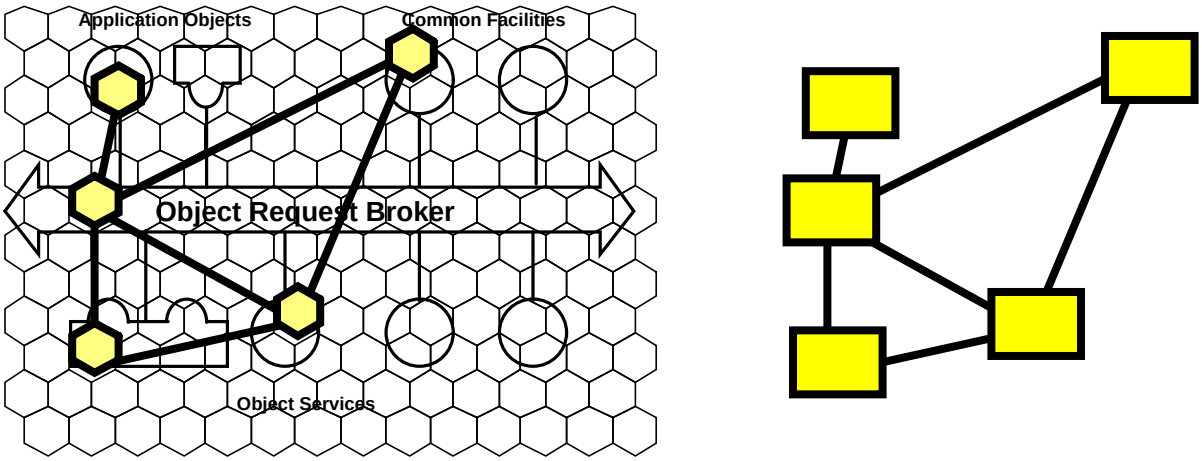

Figure 29. The role model describes a system of interacting objects

The utility for practical modeling is that a role model captures the synergy caused by a particular pattern of interacting objects. This is useful for describing a number of different phenomena such as the processes in the enterprise; the enterprise in the context of its environment; the overall enterprise information system; the tasks of individuals and their computer tools; and the enterprise domain services. It is also useful for creating generic descriptions in all these areas, descriptions that capture essential properties and that can be specialized for a variety of purposes.

> The **role abstraction** focuses on how structures of collaborating objects realize certain functionality, called **activities** as follows: An activity is a task carried out by a set of collaborating objects. In an instance of a **role model**, each **role** represents a single object involved in one or more activities. The role only describes those object properties that are of relevance to these activities. [Egil Andersen, private communication].

The role model reflects the identity of the objects of its instances. It can, therefore, precisely describe the dynamic properties of the modeled system.

We define different systems to match different *areas of concern.* A *Use Case* describes a service offered by the described system to its environment. The realization of a use case can be described by one or more scenarios. It can be described more precisely by a set of role models with role responsibilities and scenarios. Even more precisely, it can be described by a set of Labeled Transition Systems (LTS), one for each role. These LTSs should include message sends as well as message receives as their events. It will then be possible to check the overall consistency of the LTSs within the described system, and also create a summary LTS describing its overall behavior.

The role model can also support other descriptions of the system semantics, the role identity property ensures that any argument that can be applied to a system of interacting objects can also be applied to the corresponding role model.

We have said that in an instance of a role model, each role represents one specific object. The converse is not true; an object can play several roles.

This principle is used for the systematic composition of larger models from simpler ones:

> In a **synthesis operation**, we specify that objects shall play multiple roles so that all the static and dynamic properties of a **base model** are inherited into a **derived model**.

We now recast the claims of the introduction in terms of role modeling:

1. System models are to be represented as systems of objects. *A role model provides a partial description of the collaborating objects in the context of the chosen area of concern.*
2. The system model itself is a system objects. *Its* model, called the *meta-model*, is expressed in the same manner as all other models. *Different aspects such as interfaces, role models, and classes are chosen as different areas of concern and can be modeled separately. The models are brought together into a coherent whole partly through synthesis and partly through specifying classes of objects that play multiple roles.*
3. Models are observable through the ORB, making them generally available to different kinds of clients. *The meta-model is purely object oriented and therefore amenable to be managed by CORBA.*
4. The clients of system models are described in the RFP-1 as being various OA&D tools. We believe there are other, potentially more valuable, uses of a meta-model supported by a model service facility:
   - We add clients of application systems as an important extension. By giving a client access to the application system model, this client can adapt to the specific characteristics of that model. It will also be feasible to provide automatic generation of at least some parts of the client programs. *Client program developers use the published application model as base models; synthesizing them into the design model of the client program.*
   - OMG and other relevant industry bodies can publish recommended object structures in the form of system models so that developers can specialize them as needed and implement them. *Developers synthesize the recommended models into their system design models.*
   - Software vendors can offer ensembles of classes that implement the recommended structures so that application developers can derive their solutions from these object frameworks. *The software vendors offer a coordinated combination of base models for design and corresponding classes for subclassing.*

- Application developers derive their designs by specializing the recommended object structures, and possibly also derive their implementation classes from the available framework classes. They then create and publish a model of their application so that clients can adapt to it. *Application developers use the base models and classes; create their design models and also the simplified external models to be offered to the clients of the application.*

5. The meta-model supports a complete set of abstractions without mandating any of them. This permits the modeler to choose the particular set of abstractions that best communicate the ideas that are important in the given context. *A standard can mandate isolated interfaces, role models defining collaboration structures, object types, or even implementation classes.*

2.3.2 Conceptual Model

2.3.2.1 Functional subsystem

A **use case** describes the responsibilities and externally visible actions of the system under design and the actors in its environment, in pursuing a goal of an actor in the environment. A use case can have one of several outcomes: the actor is satisfied, partially satisfied, or abandoned.

Use cases do not only describe externally visible actions on the system's part. The responsibilities of a system are not always externally visible, but are what is necessary to move the system to a recognized (e.g. modeled state) which is expressed in terms of goals or partially achieved goals. A use case consists of

1. the goal statement
2. operating conditions (context in which the use case happens)
3. outcomes definition
4. one or more scenarios

A *scenario* describes the responsibilities and externally visible actions of the system under design and actors in its environment, in pursuing a goal of an actor in the environment, under a particular, stated set of circumstances giving exactly one outcome. The goal is either satisfied, partially satisfied, or abandoned.

A scenario consist of: operating conditions (conditions under which the scenario occurs), outcome (goal delivery, partial delivery, abandonment), the responsibilities, and externally visible actions of each participant (expressed as goal statements).

Use cases and scenarios are mutually recursive: each contains a set of the other. The use case is described by scenarios. Each scenario contains a set of steps; each step is phrased as a goal and therefore is a "smaller" or subordinate use case (with one of several outcomes). The conditions of the scenario are constructed so that only one of the possible outcomes of the subordinate use case can happen in that scenario. Other scenario pick up the other outcomes.

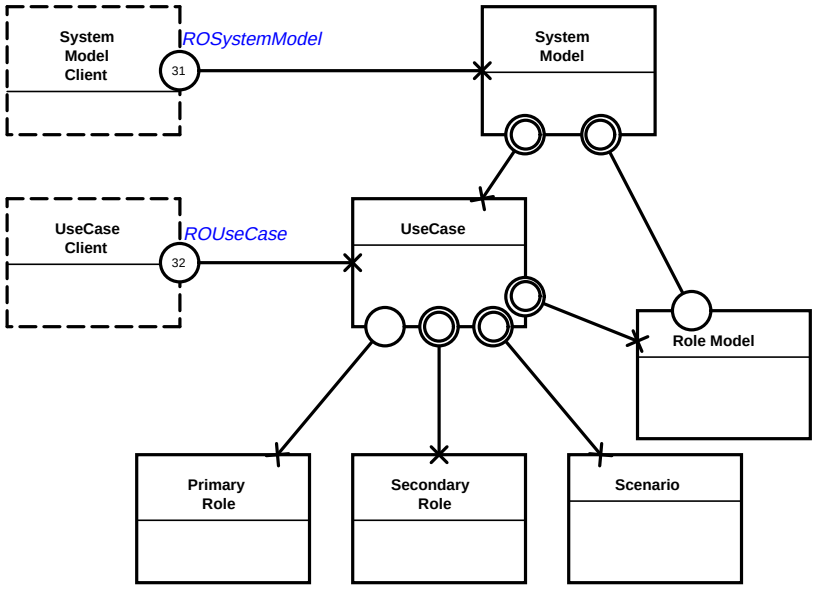

Figure 30. The Use Case meta-model

Role responsibilities

1. *SystemModelClient.* A client object wanting access to the system model.
2. *UseCaseClient*. A client wanting access to the use case model.
3. *SystemModel.* The system model subsystem.
4. *UseCase.* An object playing this role is responsible for managing the actions needed to fulfill a goal. It should also know any constraints such as constraints on the sequencing of individual operations.
5. *PrimaryRole*. The actor responsible for maintaining user goals.
6. *SecondaryRoles*. The actors responsible for supporting the use case functionality.
7. *Scenario.* Responsible for managing an execution example in the form of a sequence of interactions.
8. *RoleModel.* Responsible for representing the pattern of objects that can execute the use case.

The following paragraphs expand upon the definitions above:

A *primary actor* is one having a goal requiring the assistance of the system. A *secondary actor* is one from which the system needs assistance to satisfy its goal. Secondary actors are played by the roles appearing in the scenarios. One of the actors is designated as the system under discussion.

Each actor has a set of *responsibilities*. To carry out that responsibility, it sets some goals. To reach a goal, it performs some actions. An *action* is the triggering of an interaction with another actor, calling upon one of the responsibilities of the other actor. If the actor delivers, then the primary actor is closer to reaching its goal.

An *external actor* can be a person, a group of people, or a system of any kind. It will in general be represented by an environment role. The internal actor may be the system in design, a subsystem or an object. It will be represented by a system role. The system in design consists of subsystems, which consist of objects. Actors have behavior(s). The top-level behavior is a responsibility. It contains goals, which contain actions. An action triggers an interaction. The interaction is one actor's goal calling upon another actor's (or its own) responsibility. If a secondary actor does not deliver its responsibility, for whatever reason, the system under design or the primary actor has to define a way to get the goal satisfied. This communication model operates at all scales, from organization to individual objects.

Since Use Cases can be decomposed, it is quite possible to have Use Cases as 'Actors' to other use cases. Our definition of Primary Actor still applies.

Use Cases need not be Actor initiated. (How else could you describe the a system's response of performing a time-based activity- like a batch process?). Modeling Time as an actor is awkward and unnatural, but still time passing is a detectable event causing a Use Case to execute.

The use case is further factored by distributing the overall task to the system objects. This is done by assigning responsibilities to the system roles. Formally, the responsibility is represented by the interfaces showing the operations that must be supported by the role. Textually, the responsibility can be recorded as a *named_characteristic* in the appropriate element.

Notions that are central to responsibility driven design are the following:

1. An object has public and private responsibilities. Public responsibilities are recorded as described above. Private responsibilities can be recorded in a path to the role itself (a path to self).
2. A responsibility is (from an outside perspective) an obligation to answer a question or to perform an operation.
3. Responsibilities can be further factored into message signatures. A responsibility may be supported by more than one message. It may be supported by more than one interface.
4. Responsibilities may be grouped into contracts that can be supported and/or extended.

5. In an inheritance hierarchy, classes should support the contracts defined by their superclasses. Role model synthesis supports coordinated inheritance of an ensemble of responsibilities.
6. Responsibilities are mapped onto roles. Roles are mapped onto interfaces, and thence types. Types are mapped onto classes.

In summary, responsibilities are a way of organizing a functional view of an object, role, components or interface.

2.3.2.2 Role Model subsystem

A role model describes systems of interacting objects. The extension of a role model is a set of similar systems of interacting objects.

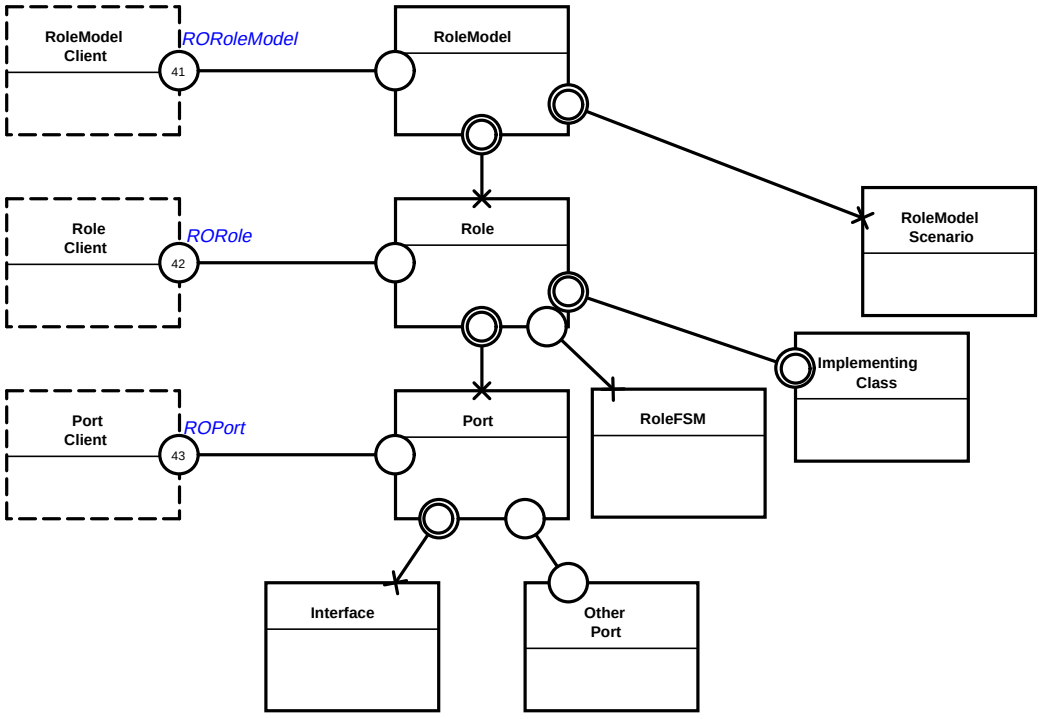

Figure 31. RoleModel collaboration view

Role responsibilities

1. *RoleModelClient.* This environment role represents objects that has reference to the role model.

2. *RoleClient.* Represents a client object having reference to a role.
3. *PortClient.* Represents a client object having reference to a port.
4. *RoleModel*. Represents the root of a role model subsystem. An instance of a role model is a structure of objects that collaborate as specified by the model.
5. *RoleModelScenario.* Represents a sample activity execution in the form of a sequence of role interactions. The scenario is detailed in section 2.3.2.3 below.
6. *Implementing class.* Reference to a class that implements this role.
7. *Role*. Represents one specific object in each instance of the role model.
8. *RoleFSM.* The behavior of the role may be defined in terms of a Finite State Machine associated with the role. The events of the FSM are message sends and receives, in addition to internal events that are caused by object actions that are invisible in this role model.
9. *Port.* A Port is an abstraction on a variable, permitting the object represented by the adjoining role to execute operations in the object represented by *OtherRole*.
10. *OtherPort.* In an instance of a role model, *OtherPort* represents the port's opposite on the communication path.
11. *Interface.* The *Interfaces* define the operations that may be executed through the adjoining *Port*. An object represented by *Role* may, but is not required to, invoke the specified operations. The object represented by *OtherRole* must implement methods for all the specified operations.

A **contract** could be associated with a Port. A contract is more than an interface which can be bound to a role or class. It can be abstracted into responsibilities and obligations (on both the server and the client). You can describe a 'protocol' between client and server roles (indicating sequence) as well as the 'fine print or contractual obligations' - e.g. Bertrand Meyer's pre- and post-conditions.

2.3.2.3 Role model Scenario

A scenario describes a sample execution sequence.

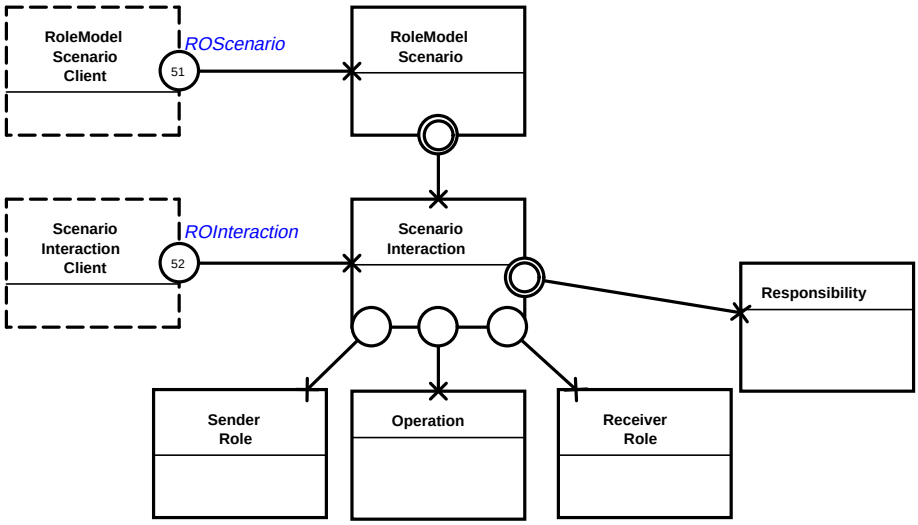

Figure 32. Scenario collaboration view

Role responsibilities

1. *RoleModelScenarioClient.* This role represents an object having reference to a Scenario.
2. *ScenarioInteractionClient.*This role represents an object having reference to a Scenario interaction.
3. *RoleModelScenario.* This role represents an object holding a sample execution interaction sequence.
4. *ScenarioInteraction.* A ScenarioInteraction describes that SenderRole invokes the specified operation in ReceiverRole.
5. *SenderRole.* Represents the sender object in a particular interaction.
6. *Operation.* Represents the operation invoked in a particular interaction.
7. *ReceiverRole.* Represents the receiver object in a particular interaction.
8. *Responsibility.* The associated interaction is a particular fulfillment of this Responsibility.

2.3.2.4 Finite State Machine (RoleFSM)

This subsystem represents a Labeled Transition System describing the aspects of the object behavior that are relevant to the current area of concern.

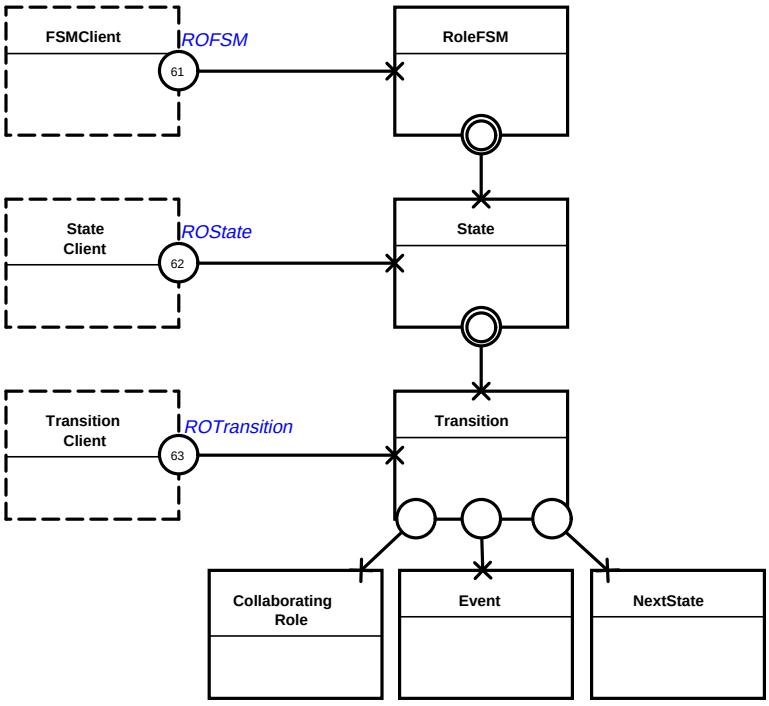

Figure 33. FSM collaboration view

Role responsibilities

1. *FSMClient.* Represents an object having a reference to an FSM object.
2. *StateClient.* Represents an object having a reference to an FSM State object.
3. *TransitionClient.* Represents an object having a reference to an FSM transition.
4. *RoleFSM.* Represents the holder of an FSM description of role behavior.
5. *State.* Represents a state in the role's behavior description.
6. *Transition.* Represents a state transition description.
7. *Operation.* Represents the event that triggers the transition. This event can be either the reception of a message from a collaborator, the sending of a message to a collaborator, or it can be an internal event in the described role.
8. *CollaboratingRole.* Represents the sender of a received message or the receiver of a sent message. Undefined if the operation is internal to the role.
9. *NextState.* Represents the state reached after the transition.

2.3.2.5 Role Model Synthesis

A SynthesisMap maps a base role model with its roles and ports onto corresponding elements in a derived model.

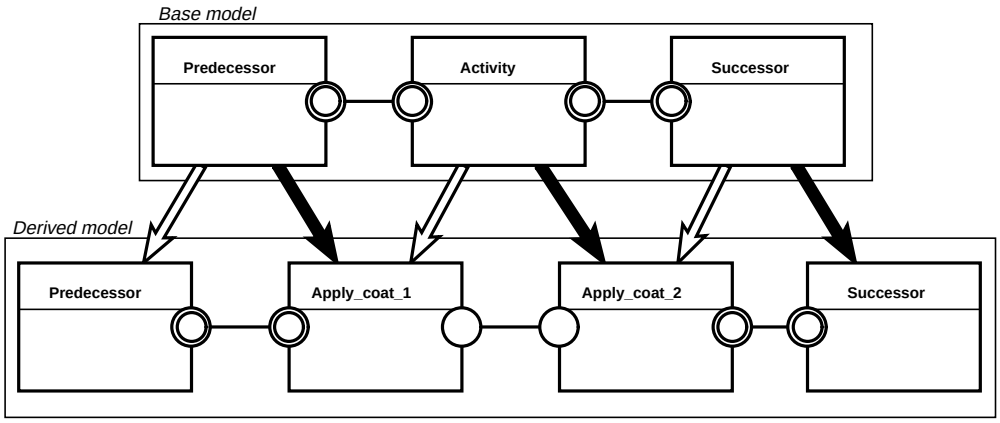

Figure 34. Two synthesis operations

Figure 34 above illustrates two synthesis operations. The notation is shorthand for the two synthesis operations that are detailed in figures 35 and 36 below.

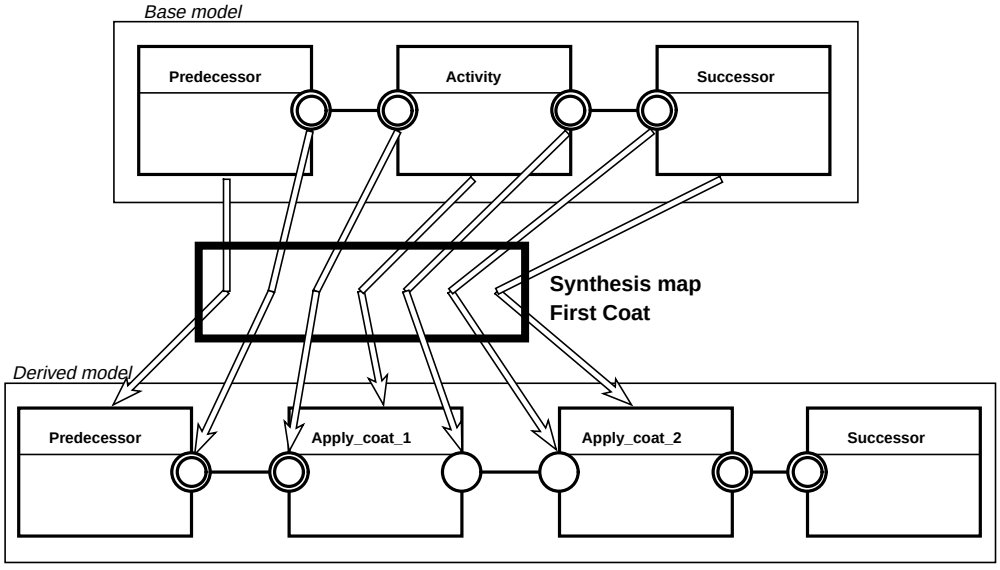

Figure 35. The details of the synthesis operation for the first coat of paint

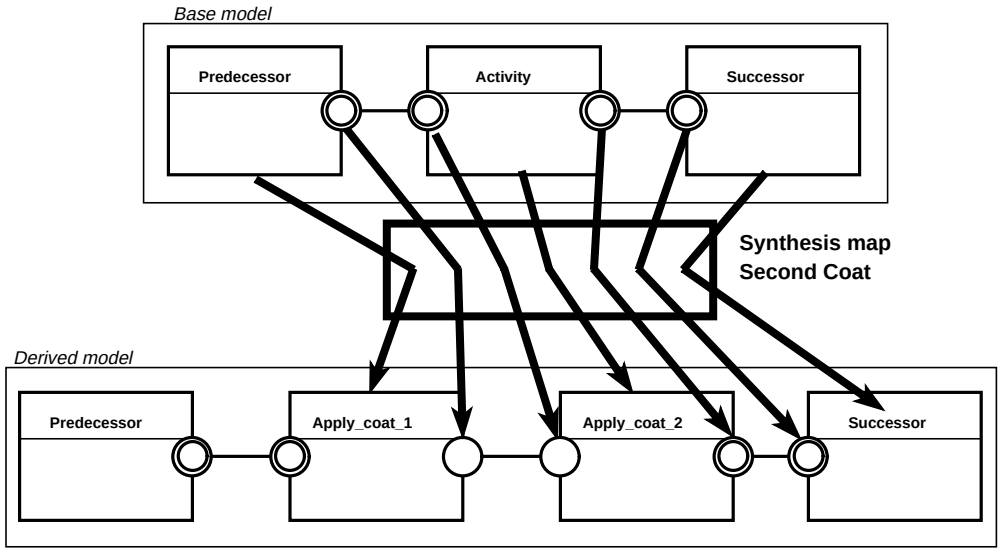

Figure 36. The details of the synthesis operation for the second coat of paint

A synthesis relationship is represented by a SynthesisMap that maps a derived Role Model with its Roles and Ports and other elements onto the corresponding elements in a base Role Model. The collaboration view of figure 37 shows the structure of this powerful mechanism. The *ROBaseElement* interface enables navigation from a element to its derived elements. The *RODerivedElement* interface enables navigation from a element to its base elements.

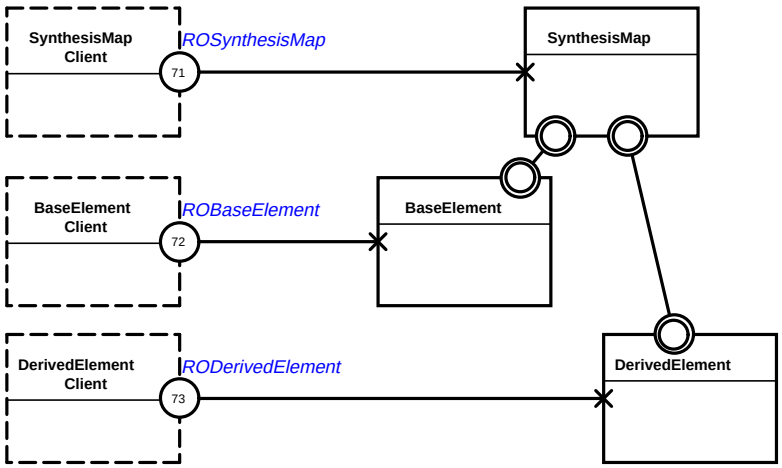

Figure 37. The SynthesisMap

Role responsibilities

1. *SynthesisMapClient.* Represents an object that needs to examine the map between base and corresponding derived elements.
2. *BaseElementClient.* Represents an object that has reference to a base element and needs to reference its synthesis map.
3. *DerivedElementClient.* Represents an object that has reference to a derived element and needs to reference its synthesis map.
4. *SynthesisMap.* A SynthesisMap maps a base role model and all its elements onto a derived model and its corresponding elements.
5. *BaseElement.* The SynthesisMap holds a reference to the base model and all its elements.
6. *DerivedElement.* The SynthesisMap holds a reference to the derived model and all its elements.

Comparing figure 37 with the two detailed illustrations in figures 35 and 36, we see that the role model captures the essence of the synthesis relationship without cluttering details. Figure 37 does not show that all *RoleModel, Role,* and *Port* objects are also capable of playing the roles of *BaseElement* and *DerivedElement.* Since these are common properties, they are best expressed in the interface and class hierarchies. Figure 38 shows a possible type or class hierarchy, the corresponding interface hierarchy is as follows:

```
interface ROBaseElement : ROElement { /* Port # 72 */
  sequence<ROSynthesisMap> derived_synthesis_maps();
  sequence<RORMElement> derived_elements();
  };

interface RODerivedElement : ROElement { /* Port # 73 */
  sequence<ROSynthesisMap> base_synthesis_maps();
  sequence<RORMElement> base_elements();
  };

interface RORMElement : ROBaseElement , RODerivedElement {
  };

interface ROSynthesisMap : ROElement { /* Port # 71 */
  sequence<RORMElement> base_elements_at(in RORMElement
derived_element);
  sequence<RORMElement> derived_elements_at(in RORMElement
base_element);
  };
```


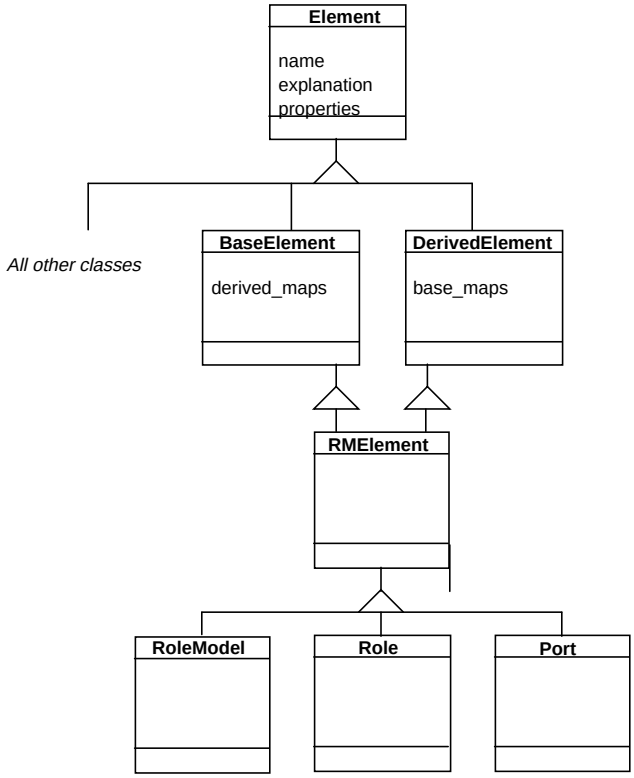

Figure 38. Class hierarchy for elements supporting synthesis

2.3.3 Complete System Model Interface Descriptions

```
/* Synthesis base types */

interface ROBaseElement : ROElement { /* Port # 72 */
  sequence<ROSynthesisMap> derived_synthesis_maps();
  sequence<RORMElement> derived_elements();
  };

interface RODerivedElement : ROElement { /* Port # 73 */
  sequence<ROSynthesisMap> base_synthesis_maps();
  sequence<RORMElement> base_elements();
  };

interface RORMElement : ROBaseElement , RODerivedElement {
  };

interface ROSynthesisMap : ROElement { /* Port # 71 */
  sequence<RORMElement> base_elements_at(in RORMElement
derived_element);
```



```
  sequence<RORMElement> derived_elements_at(in RORMElement
base_element);
  };

/* Functional model */

interface ROSystemModel : ROElement {
  sequence <ROUseCase> use_cases();
  sequence <RORoleModel> role_models();
  };

interface ROUseCase : ROElement { /* Port # 31 */
  string goals();
  RORole primary_role();
  sequence <RORole> secondary_roles();
  sequence <ROScenario> scenarios();
  sequence <RORoleModel> role_models();
  };

interface ROResponsibility : ROInterfaceElement {
  sequence <ROOperation> resp_operations();
  };

/* Core role model */

interface RORoleModel : RORMElement { /* Port # 41 */
  sequence<RORole> roles();
  sequence<ROScenario> scenarios();
  };

interface RORole : RORMElement { /* Port # 42 */
  sequence<ROPort> ports();
  ROFsm role_fsm(out boolean hasFsm);
  sequence<ROClass> implementing_classes();
  sequence <ROResponsibility> responsibilities();
  };

interface ROPort : RORMElement { /* Port # 43 */
  sequence<ROInterface> interfaces();
  RORole own_role();
  ROPort other_port();
  RORole other_role();
  };

/* Role Scenario */

interface ROScenario : ROElement { /* Port # 51 */
  sequence<ROInteraction> interactions();
  };

interface ROInteraction : ROElement { /* Port # 52 */
  ROOperation invoked_operation();
  sequence <ROResponsibility> responsibilities();
  RORole sender_role();
  RORole receiver_role();
  };

/* FSM */
```

```
/*  NOTE this is an extended FSM where the messages an object sends
are part of the picture. */

interface ROFsm : ROElement { /* Port # 61 */
  sequence<ROState> states();
  };

interface ROState : ROElement { /* Port # 62 */
  sequence<ROTransition> transitions();
  };

enum Direction {Receive, Send};

interface ROTransition : ROElement { /* Port # 63 */
  Direction msg_direction();
  RORole this_role();
  RORole collaborator_role();
  ROState next_state();
  ROOperation event();  /* operation receive or send */
  };
```

2.4 Program-Centered Meta-Model

2.4.1 Rationale

The Type abstraction focuses on the external properties of the object, i.e., its implemented set of operations. The type is illustrated in figure 39. Types are commonly organized in a *type hierarchy*, where a *subtype* implements all the operations of its *supertype* and adds some of its own. Since the type abstraction hides implementation details, it is useful for describing the component parts of distributed systems.

> **type**: The specification of an interface and a collection of objects to which the interface applies. Also known as *object type.* **Contrast with *class.***

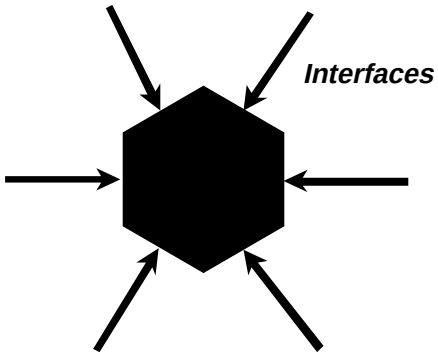

Figure 39. The Type abstraction defines external object properties and hides internal implementation

There is a many-to-many relationship between role and type. A type can specify objects that are capable of playing several roles; and a role can be played by several types of objects.

A type can be specified explicitly as a set of interfaces. It can also be specified more precisely by the set of roles that its instances shall be capable of playing. For example, the Type *Activity* can be defined to support the capabilities of the set of roles *{Predecessor, Activity, Successor}.* Implicit in this specification is that the Type must implement the interfaces associated with the relevant ports. But the Type specification can have additional information: instances of this type will have variables implementing the designated ports. Some of these ports may be overlapping, since a collaborator object can play several roles.

The class abstraction focuses on the internal implementation of the object as illustrated in figure 40. In programming, a class denotes a program that controls the properties of a set of objects, called the *instances* of the class. Classes are commonly organized in a *class hierarchy*, where a *subclass* inherits all the code of its *superclass*, and where the subclass may modify the methods of its superclass and also add new methods and object variables. The class supports reusable code and is useful in programming for code sharing. It is typically the foundation for reusable class libraries.

> **class**: A definition of an implementation (methods and data structures) shared by a group of objects. A template for defining the methods and variables for a particular type of object. All objects of a given class are identical in form and behavior but contain different data in their variables. **Contrast with *type.***

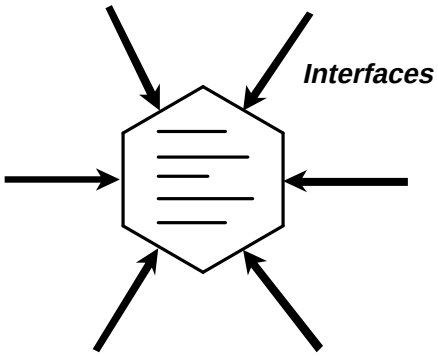

Figure 40. The Class abstraction defines external object properties together with its internal implementation

Different classes can implement the same types, and a class can satisfy the requirements of several types. A class can define objects that can play many roles; and a role can be played by objects of different classes.

The type inheritance relationships hints at a possible class inheritance relationship. But these inheritance relationships need not always coincide since type inheritance is used to factor out common interfaces while class inheritance is used to factor out common code.

2.4.2 Conceptual Model

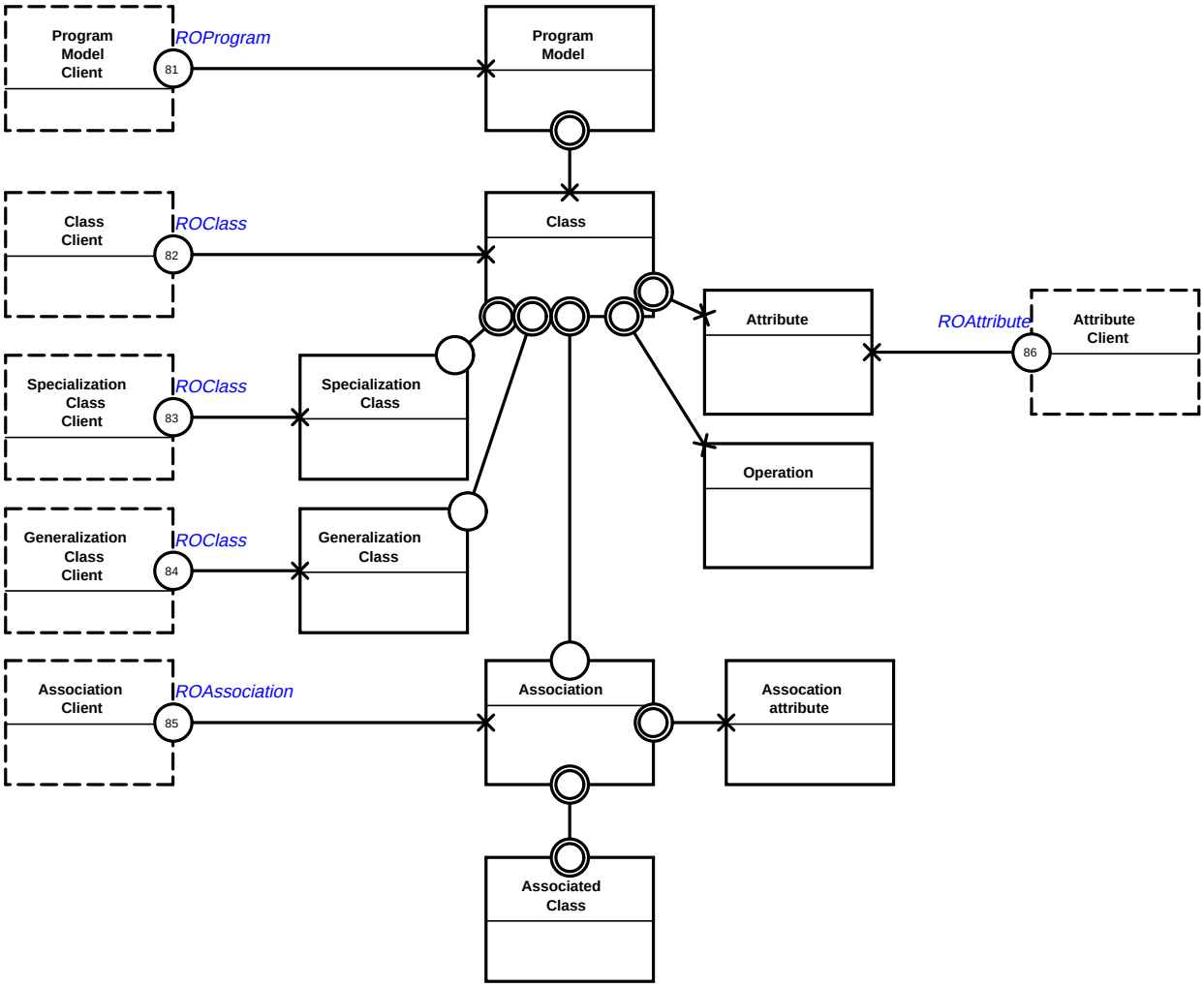

Figure 41. Class collaboration diagram

This model is supposed to give a taste of UML. It is intended to be replaced by the full UML model (assuming that the UML meta-model will be object oriented). The role responsibilities are as follows:

1. *ProgramModelClient.* Object having access to a ProgramCenteredMetaModel.
2. *ClassClient.* Class Client role represents a client that wants to access a class, its properties and associations
3. *SpecializationClassClient.* Represents an object that has access to a superclass and wants to find its subclasses.

4. *GeneralizationClassClient.* Represents an object that has access to a subclass and wants to find its superclasses.
5. *AssociationClient.* Represents an object that has access to an association.
6. *AttributeClient.* Represents an object that has access to an attribute.
7. *ProgramModel.* Represents the program-centered dimension of the meta-model.
8. *Class.* The Class Role represents a model class element
9. *SpecializationClass.* Represents a subclass.
10. *GeneralizationClass.* Represents a superclass.
11. *Attribute.* Represents the instance variables of a class
12. *Operation.* Represents a method in the class
13. *Association.* A Relationship Association maintains a relationship between two or more Classes.
14. *AssocationAttribute.* Represents an attribute related to an association.
15. *AssociatedClass.* Represents a class that is associated with to the *Class*.

2.4.3 Interface Descriptions

```
interface ROProgram : ROElement { /* Port #81 */
  sequence<ROClass> classes();
  };

enum ClassKind {SimpleClass, TemplateClass, InstantiatedClass};
enum OverrideKind {Virtual, Leaf, Deferred, Extensible};

interface ROClass : ROElement { /* Port #82, 83, 84 */
  ClassKind class_kind();
  OverrideKind override_kind();
  boolean is_utility();
  sequence<RORole> implemented_roles();
  sequence<ROInterface> implemented_interface();
  sequence<ROAttribute> attributes();
  sequence<ROOperation> operations();
  sequence<ROClass> generalization_classes();
  sequence<ROClass> specialization_classes();
  sequence<ROAssociation> associations();
  };

enum PropertyScope {InstanceScope, ClassScope};

enum PropertyVisibility {ROPublic, ROProtected, ROPrivate};

interface ROProperty : ROElement  { /* Port #4
```

```
  PropertyScope scope();
  PropertyVisibility visibility();
  };

interface ROAttribute : ROProperty {  /* Port #86 */
  };

interface ROAssociation : ROElement { /* Port #85 */
  sequence<ROClass> participating_classes();
  sequence<ROAssociationAttribute> association_attributes();
  };

interface ROAssociationAttribute : ROInterfaceElement {
  ROType attribute_type();
  };
```

2.5 Meta-model interface specification

```
module ROMetaModel {

/*****************************************************/
/* Mention all interfaces to relax sequencing rules. */
/*****************************************************/

/* Prefix RO = ReadOnly */

 interface ROApplicationObject;
 interface ROAssociation;
 interface ROAssociationAttribute;
 interface ROAttribute;
 interface ROBaseElement;
 interface ROClass;
 interface ROClassModel;
 interface ROConst;
 interface RODerivedElement;
 interface ROElement;
 interface ROException;
 interface ROFsm;
 interface ROInteraction;
 interface ROInterface;
 interface ROInterfaceSpecification;
 interface ROMethod;
 interface ROModel;
 interface ROModule;
 interface ROOperation;
 interface ROParameter;
 interface ROPort;
 interface ROProgram;
 interface ROProperty;
 interface ROResponsibility;
 interface RORMElement;
 interface RORole;
 interface RORoleModel;
 interface ROScenario;
 interface ROState;
 interface ROSynthesisMap;
 interface ROSystemModel;
 interface ROTransition;
 interface ROType;
 interface ROUseCase;
```

```
/* Common base interfaces */
/**************************/

struct DictionaryData {
  string keyword;
  string value;
  };

typedef sequence<DictionaryData> Dictionary;

interface ROElement {
  string element_name();
  string element_explanation();
  Dictionary named_characteristics();
  };

/* Conceptual metamodel - model overview */
/*****************************************/

interface ROApplicationObject { /* Port # 11 */
  ROModel model(out boolean has_model);
  };

interface ROModel : ROElement { /* Port # 12 */
  sequence<ROInterface> interfaces();
  sequence<ROSystemModel> system_models();
  sequence<ROProgram> program_models();
  };

/* Interface subsystem */
/***********************/

interface ROInterfaceElement : ROElement {
  string definition();
  };

interface ROInterfaceSpecification : ROInterfaceElement { /* Port # 21
*/
  sequence<ROModule> int_modules();
  sequence<ROInterface> int_interfaces();
  sequence<ROType> int_types();
  sequence<ROConst> int_constants();
  sequence<ROException> int_exceptions();
  };

interface ROModule : ROInterfaceElement { /* Port # 22 */
  sequence<ROInterfaceSpecification> scoped_interface_specification();
  };

interface ROInterface : ROInterfaceElement { /* Port # 23 */
  sequence<ROOperation> int_operations();
  sequence<ROResponsibility> int_responsibilities();
  sequence<ROInterface> int_base_interfaces();
  sequence<ROType> int_types();
  sequence<ROConst> int_constants();
  sequence<ROException> int_exceptions();
```



```
  sequence<ROPort> using_ports();
  sequence<ROClass> implementing_classes();
  };

enum InvocationSemantics {Synchronous, Asynchronous,
SynchronousDeferred};

interface ROOperation : ROInterfaceElement { /* Port # 24 */
  boolean op_is_oneway();
  ROType op_return_type();
  sequence<ROParameter> op_parameters();
  InvocationSemantics semantics();
  };

interface ROType : ROInterfaceElement {
  };

interface ROConst : ROInterfaceElement {
  };

interface ROException : ROInterfaceElement {
  };

enum ParameterDirection {direction_in, direction_out,
direction_inout};

interface ROParameter : ROInterfaceElement { /* Port # 25 */
  ParameterDirection parameter_direction();
  ROType parameter_type();
  };

/*  System Model */
/****************/

/* Synthesis base types */

interface ROBaseElement : ROElement { /* Port # 72 */
  sequence<ROSynthesisMap> derived_synthesis_maps();
  sequence<RORMElement> derived_elements();
  };

interface RODerivedElement : ROElement { /* Port # 73 */
  sequence<ROSynthesisMap> base_synthesis_maps();
  sequence<RORMElement> base_elements();
  };

interface RORMElement : ROBaseElement , RODerivedElement {
  };

interface ROSynthesisMap : ROElement { /* Port # 71 */
  sequence<RORMElement> base_elements_at(in RORMElement
derived_element);
  sequence<RORMElement> derived_elements_at(in RORMElement
base_element);
  };

/* Functional model */
```

```
interface ROSystemModel : ROElement {
  sequence <ROUseCase> use_cases();
  sequence <RORoleModel> role_models();
  };

interface ROUseCase : ROElement { /* Port # 31 */
  string goals();
  RORole primary_role();
  sequence <RORole> secondary_roles();
  sequence <ROScenario> scenarios();
  sequence <RORoleModel> role_models();
  };

interface ROResponsibility : ROInterfaceElement {
  sequence <ROOperation> resp_operations();
  };

/* Core role model */

interface RORoleModel : RORMElement { /* Port # 41 */
  sequence<RORole> roles();
  sequence<ROScenario> scenarios();
  };

interface RORole : RORMElement { /* Port # 42 */
  sequence<ROPort> ports();
  ROFsm role_fsm(out boolean hasFsm);
  sequence<ROClass> implementing_classes();
  sequence <ROResponsibility> responsibilities();
  };

interface ROPort : RORMElement { /* Port # 43 */
  sequence<ROInterface> interfaces();
  RORole own_role();
  ROPort other_port();
  RORole other_role();
  };

/* Role Scenario */

interface ROScenario : ROElement { /* Port # 51 */
  sequence<ROInteraction> interactions();
  };

interface ROInteraction : ROElement { /* Port # 52 */
  ROOperation invoked_operation();
  sequence <ROResponsibility> responsibilities();
  RORole sender_role();
  RORole receiver_role();
  };

/* FSM */
/*  NOTE this is an extended FSM where the messages an object sends
are part of the picture. */

interface ROFsm : ROElement { /* Port # 61 */
  sequence<ROState> states();
  };
```

```
interface ROState : ROElement { /* Port # 62 */
  sequence<ROTransition> transitions();
  };

enum Direction {Receive, Send};

interface ROTransition : ROElement { /* Port # 63 */
  Direction msg_direction();
  RORole this_role();
  RORole collaborator_role();
  ROState next_state();
  ROOperation event();  /* operation receive or send */
  };

/*  Program Model */
/*****************/

interface ROProgram : ROElement { /* Port #81 */
  sequence<ROClass> classes();
  };

enum ClassKind {SimpleClass, TemplateClass, InstantiatedClass};
enum OverrideKind {Virtual, Leaf, Deferred, Extensible};

interface ROClass : ROElement { /* Port #82, 83, 84 */
  ClassKind class_kind();
  OverrideKind override_kind();
  boolean is_utility();
  sequence<RORole> implemented_roles();
  sequence<ROInterface> implemented_interface();
  sequence<ROAttribute> attributes();
  sequence<ROOperation> operations();
  sequence<ROClass> generalization_classes();
  sequence<ROClass> specialization_classes();
  sequence<ROAssociation> associations();
  };

enum PropertyScope {InstanceScope, ClassScope};

enum PropertyVisibility {ROPublic, ROProtected, ROPrivate};

interface ROProperty : ROElement  { /* Port #4
  PropertyScope scope();
  PropertyVisibility visibility();
  };

interface ROAttribute : ROProperty {  /* Port #86 */
  };

interface ROAssociation : ROElement { /* Port #85 */
  sequence<ROClass> participating_classes();
  sequence<ROAssociationAttribute> association_attributes();
  };

interface ROAssociationAttribute : ROInterfaceElement {
  ROType attribute_type();
  };

}; /* End-of-module */
```

2.6 Glossary

activity a set of all possible scenarios among participating objects. The activity starting operation is called a stimulus (trigger), the possibly resulting operation is called outcome operation (response).

activity aggregation A base activity is wholly included as part of a derived activity

activity superposition An activity in a base model also appear as an activity of a derived model.

aggregation (containment) an association between objects or roles showing that an object is contained in or part of another.

base model A role model that is composed into a *derived model* through synthesis.

class A definition of an implementation (methods and data structures) shared by a group of objects. A template for defining the methods and variables for a particular type of object. All objects of a given class are identical in form and behavior but contain different data in their variables. **Contrast with *type.***

derived model A role model that is composed from a *base model* through synthesis.

environment For a given system, the *environment* is the set of all objects outside the system whose actions affect it, and also those objects outside the system whose attributes are changed by its actions.

Finite State Machine (FSM) A behavioral description of an object's states and transitions.

interface An *interface* is a listing of operations and attributes that an object provides. This includes the signatures of operations, and the types of the attributes. An object *satisfies* an interface if it can be specified as a target object in each potential request described by the interface.

Interface inheritance *Interface inheritance* is defined as the construction of an interface by incremental modification of other interfaces. The IDL language provides interface inheritance.

meta-model A model of the modeling concepts used to define models.

model A simplified representation of a real-world concept used to define an application, a tool, a business, or a technology.

method The program fragment defining the effect of a message in the receiving object.

object the representation of an identifiable unit (entity) having a collection of properties and providing a set of services. It has autonomy for its properties and services (see encapsulation).

operation (often called *message*)
a communication from one object (sender) to another (recipient). It can obey certain communication disciplines, e.g. synchronous, synchronous deferred, or asynchronous. It has a possibly empty list of parameter values, it may also have a return value.

port the named part of an object enabling to connect the object to one or more collaborators, and defining the set of permitted operations to be sent or received from the object.

Program-centered approach The *Program-centered approach (PC)* focuses on the class as the central abstraction. Supporting concepts are types and various kinds of relationships and associations between classes and types.
See also **System-centered approach.**

System-centered approach. The *System-Centered approach (SC)* focuses on systems of interacting objects. Use cases, Responsibility Driven Design, collaboration diagrams, message scenarios, and system behavior models belong in this dimension. A precisely defined abstraction is the *role model*, which describes the static and dynamic properties of a structure of interacting objects in the context of a specific subject, called an *Area of Concern.*
See also **Program-centered approach.**

role an abstraction of the position and purpose of an object in a role model.

role model an abstraction of a system that highlights the statics and dynamics of a pattern of a set of interacting objects.

The *role* abstraction focuses on how structures of collaborating objects realize certain functionality, called activities as follows: An activity is a task carried out by a set of collaborating objects. In a role model, each role represents a single object involved in one or more activities. The role only describes those object properties that are of relevance to these activities. [Egil Andersen].

scenario a specific sequence of operations among objects, consisting of a stimulus to an object, cascading operations among the objects, and an optional outcome operation (response). For a given stimulus the set of all possible scenarios makes up an activity.
See also *activity*.

synthesis In a *synthesis operation*, we specify that objects shall play multiple roles so that all the static and dynamic properties of a *base model* are inherited into a *derived model.*

system A system is a part of the real world which we choose to regard as a whole, separated from the rest of the world during some period of consideration; a whole that we *choose* to consider as containing a collection of objects, each object characterized by a selected set of associated attributes and by actions which may involve itself and other objects.

type The specification of an interface and a collection of objects to which the interface applies. Also known as *object type*. **Contrast with *class.***

use case A use case describes the actors external to a system, their goals and actions, together with the system's responsibilities and externally visible actions.
See also *scenario*, which can be an instance of a use case.

Chapter 3
Resolution of technical and non-technical issues

Some of the RFP-1 requirements are met as follows:

1. *Submissions must define a meta-model that represents the semantics of OA&D methods, including one or more of the following core models: Static models, dynamic models, usage models, architectural models.*
   We have elected a different approach to overall meta-model structuring, namely *interface, role model,* and *class.* Static, dynamic, usage, and architectural properties can be described in terms of these models, but we have not done so explicitly in this proposal since the terms seem more associated with development process and specific mapping between objects and real-world entities than to object modeling as such. The meta-model defined in chapter 2 includes static and dynamic properties of classes, interfaces, and role models. The meta-model is open ended so that additional features can be added as required.
2. *Submissions must define the relationships among the models and allow for defining constraints among the model elements.*
   The relationships between the model elements are clearly defined in chapter 2. Constraints have not been included because we expect this feature to be adequately covered in other proposals.
3. *Submissions must include a formal definition of meta-model constructs. The technique for specifying this definition is left to the discretion of the Submitters.*
   No formal definitions beyond the IDL interfaces have been included here. For the interface and class portions, we expect this to be adequately covered by other submitters. The formal definition of role modeling and role synthesis is the theme of a doctoral dissertation being in the final stages of preparation at the University of Oslo. It was felt premature to include parts of this work here.
4. *Submissions must include a diagrammatic explanation of meta-model constructs, in order to help reviewers understand the submissions. The notation for specifying these diagrams are left to the discretion of the Submitters. However, if the notation is not fully defined as a response to optional requirement 5.6.1, then they must be explained here.*
   Role models and other diagrams are given in chapter 2. Some notation is specified in chapter 9.
5. *Submissions must define the interfaces that enable models or model elements created by one tool to be accessed from another tool. The model access facility may be an import facility or a connection facility or both.*
   The facility proposed is a read-only object model, i.e., an import facility. See chapter 2.1.1.

6. *Model Notation. Submissions may provide a set of notations for representing models. The notation specified in response to this optional requirement will be considered for adoption by the Object Analysis and Design TF. Submissions may provide a notation in response to this optional requirement that differs from the notation they use in response to mandatory requirement 5.5.1.*
   The proposed role model notation is given in chapter 9. Notation for interfaces should be textual and follow IDL. Notation in the class dimension should follow UML, which was still being changed at the time of this writing.
7. *Model Notation Representational Adequacy. Submissions that provide a set of notations for representing models may show how those notations are used to represent the models included in the submissions in response to 5.5.1. During the convergence period, Submitters should expect to be asked to show how their model notations represent the Meta-Object Facility meta-meta-model.*
   In chapter 2, the meta-model is presented using the meta-model concepts and notation.

Chapter 4
Specification Dependencies

This proposal is believed to be fully compatible with the spirit and specifications of the OMG.

Models conforming to the proposed OOram meta-model will be systems of objects that are available through CORBA. No specific services have been applied, but the Event Service could be used to inform a client when an object is removed from the model.

Chapter 5
Relationship to CORBA

This proposal is fully compatible with CORBA and IDL. It does not propose or assume any extensions.

The OOAD Metamodel will be stored in a repository based on the OMG Meta-Object facility, being processed within OMG in parallell with the OOAD RFP. During the time after the initial submissions, the common aspects between these two RFP's can be synchronized.

Chapter 6
Relationship to OMG Object Model

RFP-1 Requirements:

1. *Proposed extensions to OMG IDL, CORBA, Object Services, OMG Object Model shall be identified (p.11)*
   No extensions proposed.
2. *Define, explain, and/or demonstrate constraints on object behaviors. (p.13)*
   No such constraints specified.
3. *Submitters must address the relationships between the meta-model constructs that represent OA&D objects and the meta-meta-model specifications of the Meta-Object Facility. (p.17)*

OMG has issued an RFP for multiple interfaces [MIC] which indicates that CORBA will include facilities to support multiple interfaces and composition of distributed objects. The idea of objects having multiple interfaces fits very well with roles and role-modeling concepts. A system of objects are described in terms of interactions and interfaces. Each object may interact through several ports representing exporting interfaces of system objects. Methods (such as OOram) supporting role modeling are therefore well suited for modeling systems of objects having several interfaces.

Since multiple interfaces are forthcoming within the OMG standards, the OMG object analysis and design metamodel should support modeling of such concepts. This can be achieved by incorporating the semantics of roles and role-models as described in this proposal.

Chapter 7
Standards conformance

RFP-1 Requirements:

1. In accordance with Section 3.4.17 of the OMA Guide, submissions should identify their use of relevant existing industry standards or should justify why such use is not appropriate.
2. In particular, submissions should identify how their provisions relate to the architecture for system distribution defined in ISO/IEC 10746, Reference Model of Open Distributed Processes (ODP).

The OOram meta-model is well suited for supporting ODP. Taskon is a partner in ESPRIT Project Nr. 22.084: *DISGIS. DIStributed Geographical Information Systems - Models, Methods, Tools and Frameworks.* [Oldevik]. A summary of the applicability of OOram abstracted from this ongoing project:

1. *Enterprise viewpoint.* OOram is used for analysis and top level modeling.
2. *Information viewpoint.* This is essentially a data modeling viewpoint, and standard semantic modeling tools can often be appropriate. In the DISGIS project, the OOram semantic view is used here. (Semantic view is part of the OOram tools, but not part of the proposed meta-model).
3. *Computation viewpoint.* Modeling the active objects, showing the smallest distributable objects. The full power of the OOram meta-model is appropriate, the collaboration view is particularly useful.
4. *Engineering viewpoint.* System details and low level mechanisms. The OOram meta-model is not particularly relevant. An exception is the description of low level mechanisms such as distribution mechanisms and transaction mechanisms. OOram reusable models are highly appropriate for these purposes.
5. *Technology viewpoint.* Even lower level technical details. We have not found OOram appropriate for this viewpoint.

To illustrate the OOram applicability, we give some details about the mapping of ODP enterprise concepts to OOram. For a more complete discussion of this and other viewpoints, we refer to [Oldevik].

Mapping of ODP enterprise concepts to OOram
The chosen mapping between the RM-ODP Enterprise Viewpoint concepts and the OOram models, enforce a declarative (problem) specification instead of a computational (solution) one because it conforms with the enterprise viewpoint requirements in terms of languages. As a consequence the OOram models for the enterprise viewpoint use conditions (pre-, post-, invariants, etc.).

Business object concepts
The business object concepts in the enterprise viewpoint are modeled by OOram semantic view diagrams and interface view diagrams. These diagrams will be enriched with conditions.

Community
A Community is represented as an OOram semantic view role model.

Domain
Each domain is represented as a semantic role model which is the synthesis of its communities semantic role models.

Business (=information) Object
A business object is represented as an OOram role of the semantic view diagram. Conditions may describe the objects semantics, e.g. its consistency criteria.

Attribute
An attribute is represented as an OOram role attribute of the semantic view diagram. Conditions may describe the attribute semantics, e.g. constrain the set of its possible values. Attributes may also be described as simple (non-relational) properties of roles.

Operation
An operation is represented as an OOram message of the interface. Pre- and post- conditions may describe the operation semantics, e.g. the synchronization constraints when the operation is invoked (pre-conditions) and the result of invocation (post-condition).

Class of Object
The intended semantics of an object is represented by a textual description. The operational semantics of an object is represented as OOram inter-role relations of the semantic view diagram.

Coordination
The coordination concepts are modeled by OOram collaboration view, process view, scenario view and method specification view diagrams. These diagrams will be enriched with conditions.

Federation, Domain and Community
The coordination semantics in the federation, domain and community concepts is modeled by the sharing policies described in the roles which are shared between communities or/and domains.

Role
The groupware organization role concept can be modeled as an OOram role.

Actor
This concept is not directly modeled using OOram. A table should be constructed to associate actors with the roles they play.

Process
Processes are specified by OOram role models, using collaboration and scenario views The roles in the model represent groupware activities. The specification of deadlines is a problem.
The option of representing groupware activities as OOram roles allow us to have a finer specification grain which improves flexibility, the business process is specified in terms of the activities needed to accomplish its goal, "chain value". Afterwards, as can be seen in the process description these activities are synthesized in the corresponding business roles. This approach allows the perception of the business process to be independent of the business roles and of any particular company organization. Other reasons that enforced this decision were:

1. ORCHESTRA project: the need for OOram roles representing business activities was also identified [Farshchain96], *"One way to model an automatic activity in OOram might be to model a computer application as a role and then assign the automatic activity as a method of this role".*
2. Workflow: In workflow specifications activities are the basic building elements of workflow processes [WMC94].

Activity
It is specified as an OOram role. The role has two conditions which describe the activity pre- and post- conditions. These conditions can be represented in the comment area associated with the role. The specification of pre- and post- conditions is a problem.

External Activity
It is specified as an OOram environment role. It has two conditions which describe pre- and post- conditions.

Event
Events appear in pre-conditions, when the activity is triggered by an event, in post- conditions, when the activity sends an event, and in general conditions, when the events (asynchronously) interfere with the activity.

Mapping of ODP information concepts to OOram
The mapping of the RM ODP information viewpoint into OOram modeling concepts must consider the three different schema-types defined for the information viewpoint. It must also address reference points so that conformance between the enterprise and the information viewpoint specifications can be ensured. The OOram semantic view will primarily be used for modeling of the information viewpoint. This solution requires some extensions to the semantic view; attributes, cardinality constraints, static constraints on roles, and behavioral constraints.

Chapter 8
Other Information

The separation of concerns supported by role models are ideal for the description of both analysis and design patterns. The separation of interface and class as found in modern languages such as Java, is well supported in the OOram Metamodel.

Chapter 9
Model notation

We propose to follow the UML or OML notation for the class dimension. We also propose to use the OMG IDL standard for interfaces. In the following, we propose a notation for role modeling.

OOram analysis is defined as the description of some interesting phenomenon as a system of interacting objects. In data processing, *analysis* is commonly used to denote the study of what is visible to the user community, while *design* is used to denote the description of the internal construction of a new system. *OOram analysis* covers both these interpretations; we *analyze* the world as perceived by the user community, and we *analyze* the system as perceived by its creators. The antonym is **OOram synthesis**, the composition of a whole from its constituent of parts.

A **role** is an idealized object in the sense that it is an archetypical example of the object within a **role model**, and that the role's characteristics is the subset of the object's characteristics that are of interest within the limited scope of the phenomenon described by the model.

Roles represent all the properties of objects: They represent identity and attributes; they represent encapsulation, message interaction, and actions defined by **methods**. Inheritance is supported by **synthesis** as described below.

There is a many-to-many relationship between objects and roles: an object may play several different roles from the same or different role models; and a role may be played by several different objects.

When thinking about some interesting phenomenon, the OOram analyst creates an object model of the phenomenon in his or her head. This model can only be captured on paper or a computer screen as one or more **views** -- these views are different presentations of an underlying **OOram role model**. In the case of a paper report, the underlying model is abstract in the sense that it has no explicit representation. In the case of a computer-based system, the underlying model can be represented in an object database as illustrated in. figure 42.

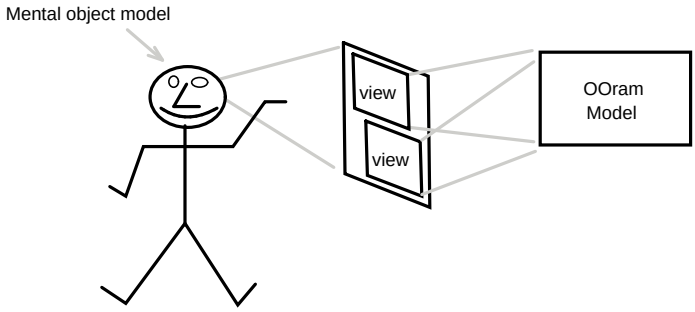

Figure 42. The analyst can see and manipulate views of an underlying model.

Systems of interacting objects can be studied in different views, with each view expressing certain aspects of the system of roles while suppressing others. The form of each view is chosen for maximum information transfer; free text, formal language, tables, and graphs are used where appropriate. The proposed meta-model supports nine different views on a role model:

1. *Area of Concern view,* which is a textual description of the phenomenon modeled by the role model. (The Area Of Concern is recorded as the explanation in the RoleModel element).
2. *Stimulus-Response view.* A tabular view showing the system's environment objects (actors), the trigger messages that start system activities, and messages conveying results to the environment roles.
3. *Role List view,* a text view showing a list of all roles with their explanations and attributes. (We do not use a tabular form, because we want to encourage comprehensive explanations).
4. *Interface view,* a formal text view showing interface definitions in the user's preferred language. IDL is taken as default.
5. *Collaboration view,* a graphical view showing the pattern of roles and the message paths between them.
6. *Scenario view,* a graphical view showing example time sequences of messages flowing between the roles.
7. *State Diagram view.* A graphical view. There may be one state diagram for each role. It describes the possible states of the role, the signals that are acceptable in each state, the action taken as a result of each signal, and the next state attained after the action is completed. The only kind of signal possible in our model is the receipt of a message.
8. *Synthesis view.* A graphical view showing the system inheritance relations between role models.
9. *Synthesis table.* A tabular view showing the mapping of base model roles to derived model roles. Mapping of the corresponding ports is implicit, all base model ports and interfaces being mapped onto the derived model.

Important notes:

1. The different views are different presentations of one and the same model for the purposes of documentation and user interaction.
2. The views are *not* orthogonal. Their mutual consistency should preferably be enforced automatically, but it is also possible to do so by manual means.

Five views are described in the following subsections; we refer to [Ree 96] for description of other view notations that we do not propose to standardize in this first step.

9.1 Area of Concern view

The *area of concern view* (figure 43) is a free text describing the phenomenon modeled by the role model. The text should indicate the purpose of the model and be sufficiently precise to enable a reader to determine which phenomena are covered by the model, and which phenomena are not covered. The description must be precise because it will be used to determine which objects and activities belong to the system and which objects and activities are outside it.

The model represents the basic activity structure in activity networks
as used in project planning an control.

An activity is an abstraction on a task. It is characterized by having a duration,
a number of predecessor activities, and a number of successor activities.
An activity can start when all its predecessors are completed.
Its earliest finish is its duration after the finish of the last predecessor.
Its successors can start any time after its finish time.

The earliest start of the network activities can be easily computed by
a front-loading operation: Start with the project start time and the initial activities,
add activity durations, and iterate through the activity successors.

Similarly, a back-loading operation computes the latest completion time for all activities
starting from the end activities and the project completion time.

Figure 43. Area of Concern example

9.2 Stimulus-Response view

Stimuli are defined as messages being sent spontaneously from an environment role to one of the system roles. The sequence of actions taken by the system is called an *activity.* The *response* is one or more messages sent from the system to one or more environment roles, or some other changes that are described as a free text such as changes to object structure or attributes. The stimulus-response relationships are shown in a table as shown in figure 44; an example is shown in figure 45.

Stimulus message	**Response messages**	**Comments**
environment role name >> message name	{environment role name << message name}...	Free text description of other results

Figure 44. Graphical notation is in the form of a table.

Stimulus message	**Response message**	**Comments**
PrimaryActor >>early_start_time	PrimaryActor << early_finish_time	The early start and finish times have been recorded in the Activity objects.

Figure 45. Stimulus-Response example.

9.3 Collaboration View

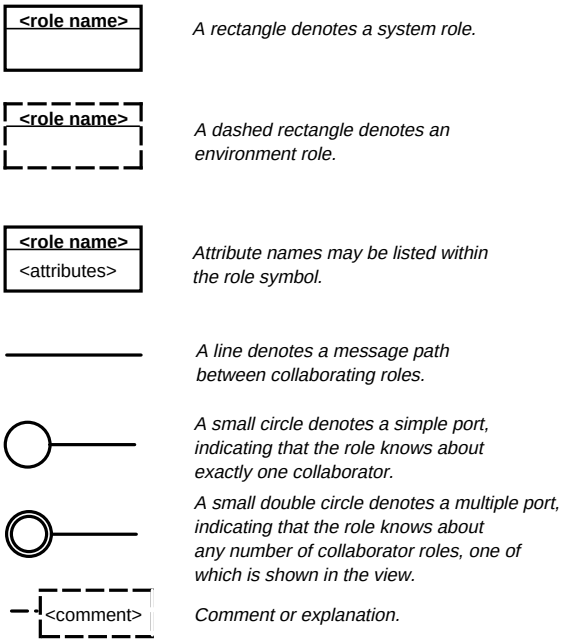

Figure 46. Collaboration view notation

The *Collaboration view* shows the roles, ports, and message paths. Our notation is shown in figure 46 and illustrated in figure 47.

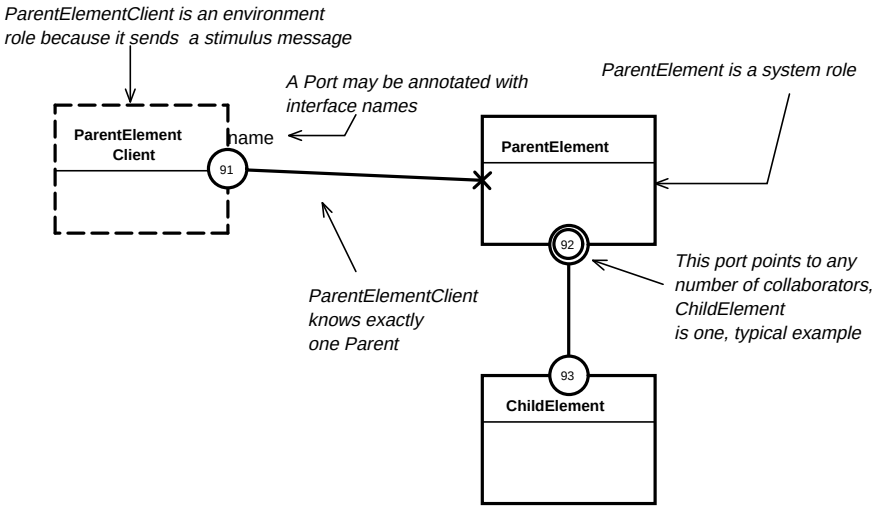

Figure 47. Collaboration view illustration

Roles may arbitrarily be lumped into *virtual roles* for convenience. A *virtual role* is a role that represents a cluster of objects rather than a single object. Virtual roles are denoted by a super-ellipse with shadow as shown in figure 48. Note that virtual roles are artifacts of the presentation and do not exist in the underlying role model.

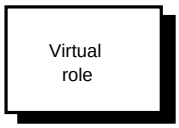

Figure 48. Virtual role notation

The *stimulus-response view* shows the system as a single virtual role together with its environment roles, as illustrated in figure 49.

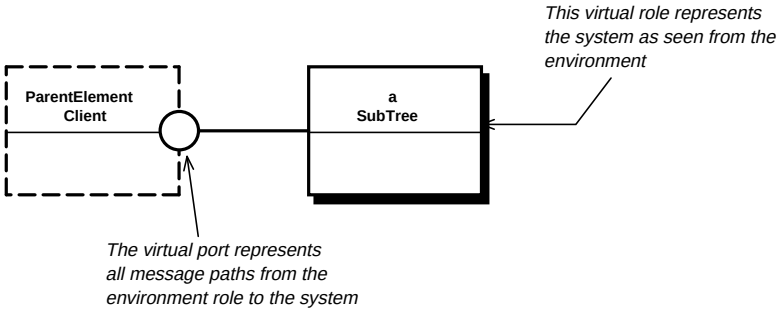

Figure 49. The stimulus-response view shows the system as a single, virtual role

Virtual roles with their associated virtual ports must be resolved into concrete roles. Figure 49 can, for example, be resolved into figure 47.

9.4 Scenario View

A *Scenario* is a description of a specific, time ordered sequence of interactions between objects.

An *interaction* represents the event of transmitting a message from a sender object to a receiver object. Both the sending and the receiving objects must be represented as roles in the role model. Interactions are strictly ordered in time and considered atomic in the context of the scenario.

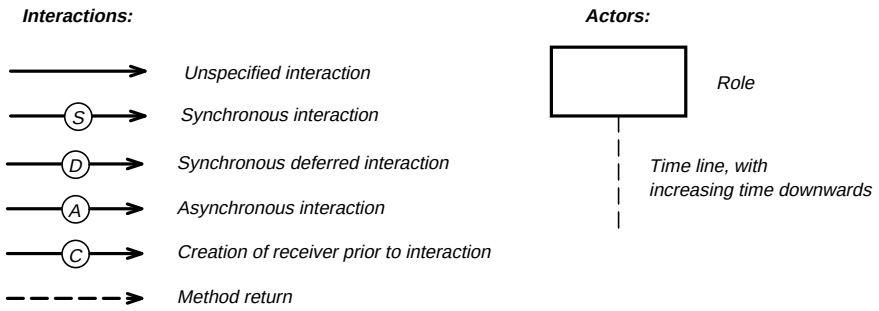

Figure 50. Scenario Notation

The Scenario thus shows an *example execution* as a sequence of messages as they flow through the structure of objects. The first message must be one of the role model's stimulus messages. The notation is shown in figure 50 and is exemplified in figure 51.

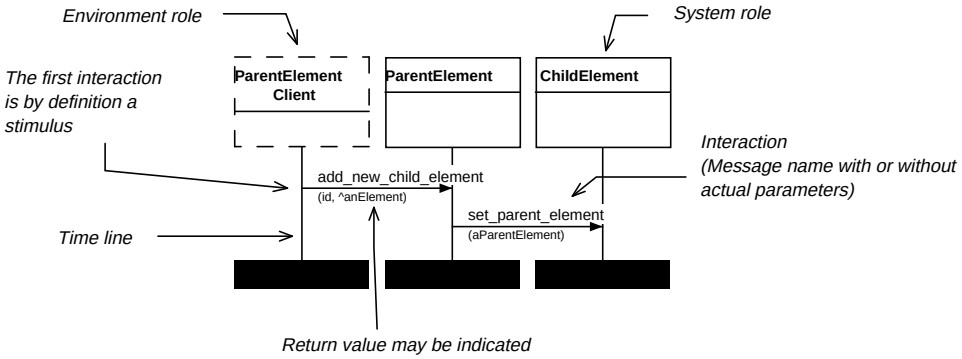

Figure 51. Scenario Illustration from element life cycle model.

In his book on object-oriented software engineering, Jacobson uses the term *actors* to denote a system's environment objects [Jacobson 92]. An OOram stimulus message is an operation initiated by an actor, and a *use case* is one or more OOram activities with their associated goals. We propose that role model scenarios are admirably suited to describe typical use case execution traces. A single scenario, as described here, will be sufficient for simple use cases, and the aggregation and synthesis operations enable us to dissect a use case down to any desired detail. Also, scenarios showing the system as a virtual role describes the actions as seen from the environment, while a scenarios showing the concrete roles illustrate how the system operates.

9.5 Notation for role model synthesis

The views described in the previous section are somehow affected by the synthesis operation. We will discuss the most important ones in the following sections.

The purpose of the inheritance views is to show the base model -- derived model relationships between role models.

In a *synthesis view*, role models are shown as rectangles. *Base model -- derived model* relationships are shown as open arrows. Derived models are shown to the right of the corresponding base models as illustrated in figure 52.

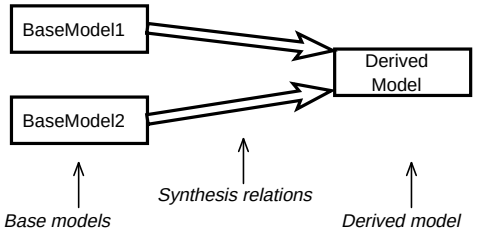

Figure 52. Synthesis view notation

An *inheritance table* is shown in table 1. It has one row for each role in the derived model. The first column shows the roles of the derived model, while the other columns show the corresponding roles in the base models. Note that *all* the roles of the base models must be accounted for in the derived model, but the reverse need not be true.

Derived model	***Base model***	***Base model***
Predecessor	*Predecessor*	
Apply-coat-1	*Activity*	*Predecessor*
Apply_coat_2	*Successor*	*Activity*
Successor		*Successor*

Table1. Example Inheritance Table: Composite Paint activity derived from basic activity model

Role model synthesis can also be illustrated in the collaboration view. Synthesis is shown as a set of equally colored open arrows going from all the base roles to the corresponding derived roles. Figure 12 shows an example with two synthesis operations.

Chapter 10
References

[Bugge91] Bugge B. et al. *Methods and tools for service creation in an intelligent network, initial document.* Kjeller, Norwegian Telecom Research, 1991 (Research Doc. No. 34/91)

[Cat] *Component-Based Development Using Catalysis / Types, Behaviors, Collaborations, Refinement, and Frameworks.* Input for OMG OOA&D Submission, Desmond Francis D'Souza and Alan Cameron Wills.

[CORBA-SER] *CORBAservices: Common Object Services Specification.* Revised edition, March 31, 1995. OMG Document Number 95-3-31.

[Farshchain96] Babak Farshchain, P.-H. Carlsen, "*Pilot 4: Workflow modelling in Norwegian BEST Pilots*", ORCHESTRA ESPRIT Project 8749, NTH/Taskon, D 5.4.3, April 1996.

[Holbæk-Hansen] Holbæk-Hansen, et.al.: *System Description and the DELTA language.* Norwegian Computing Center publication no. 523. Second printing. Oslo, 1977.

[MIC] *Multiple Interfaces and Composition*, Request for Proposal, Document ORB/96-01-04, Object Management Group.

[Nilsen] R. Nilsen, J. Simons, P. Dellafera: *Object oriented IN service provision.* TINA Workshop. L'aquila, Italy, 1993.

[Oldevik] J. Oldevik, J.O. Hove, Antonio Silva, "*Mapping between OOram, ODP, and DASCo*", Deliverable D1.3.1. Esprit project 22.084. Contact: Taskon, Oslo.

[OML] *OPEN Modeling Language (OML) Reference Manual.* Version 1.0. The OPEN Consortium 1996.

[Ree 93] Trygve Reenskaug: *The Industrial Creation of Intelligent Network Services.* TINA Workshop. L'aquila, Italy, 1993.

[Ree 96] Reenskaug, Wold, Lehne: *Working With Objects.* Manning/Prentice Hall 1996. ISBN 0-13-452930-8

[RFP-1] *Object Analysis & Design RFP-1.* OMG &/&/96

[UML] *Unified Method Metamodel guide.* Version 0.8 and 0.91Addendum. Rational Software Corporation, 1995 and September, 1996.

[Wirfs-Brock 90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener: *Designing Object-Oriented Software.* Prentice Hall, New Jersey, 1990. ISBN 0-13-629825-7

[Wirfs-Brock 95] Rebecca Wirfs-Brock, Alan McKean: *Responsibility-Driven Design.* Tutorial #44, OOPSLA '96, San Diego, California.

[WMC94] Workflow Management Coalition Members, "*Glossary - A Workflow Management Coalition Specification*", http://www.aiai.ed.ac.uk:80/WfMC/glossary.html, November 1994.

Appendix 1
Element life cycle management subsystem

App 1.1 Rationale

The meta-model described in this proposal is read-only. For completeness, we include a strategy and mechanism describing the life cycle of the model elements. This mechanism would be included in a revised proposal that included a shared, read-write model repository.

The OOram metamodel uses a strictly hierarchical object management policy. A *ParentElement* is responsible for creating its subelements, for maintaining pointers to them, and for their destruction. *ChildElements* know their *ParentElement*, and all Elements cooperate to maintain the two-way pointers.

Objects other than the object playing the ParentElement role may reference a ChildElement. These objects will be notified of the destruction of ChildElement through the standard IDL exception OBJECT_NOT_EXIST.

App 1.2 Conceptual Model

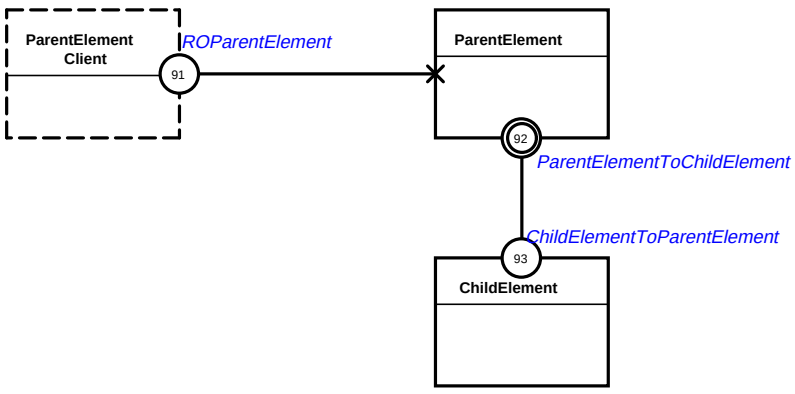

Figure 53. Object life cycle management collaboration view

Role responsibilities

1. *ParentElementClient.* This role represents any object that has reference to an object playing the *ParentElement* role and uses its services.
2. *ParentElement.* A ParentElement is responsible for managing the life cycle of a number of ChildElements.
3. *ChildElement.* This role represents an object that is managed by *ParentElement*. It is responsible for managing cross references (references not part of the basic tree structure).

App 1.3 Behavior descriptions

We show two scenarios:

1. Create and install ChildElement
2. Remove ChildElement

App 1.3.1 Create and install a ChildElement

This scenario shows the connection of a client to SubElement.

An object playing the role of ***ParentElementClient*** initiates this use case by sending the stimulus operation ***addNewChildElement:***

1. ***addNewChildElement***. Create and install a new *ChildElement*. Raise an exception if the specified *id* is not unique. This message should be duplicated and renamed in a derived model if the *ParentElement* manages *ChildElements* of different **types**.

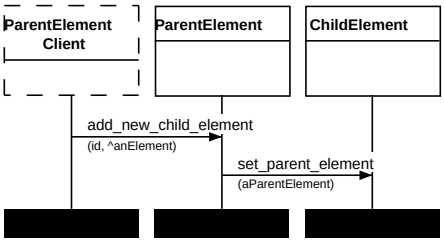

Figure 54. Create and install a ChildElement Scenario

App 1.3.2 Remove a ChildElement

This scenario shows how a client requests the removal of ChildElement and the warning of any external clients of this element.

An object playing the role of ***ParentElementClient*** commands the removal of an object playing the role of ***ChildElement***.

1. **removeChildElement**. Remove the specified *ChildElement*, raise the appropriate event for all dependents.

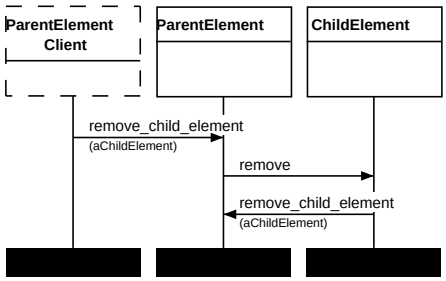

Figure 55.

App 1.4 Interface Descriptions

```
interface ROParentElement : ROElement{  /* Port #91 */
  GeneralElement add_new_child_element();
  GeneralElement remove_child_element(in GeneralElement child);
  sequence<GeneralElement> child_elements();
  };

interface ParentElementToChildElement : ROElement{  /* Port #92 */
  GeneralElement set_parent_element(in GeneralElement parent);
  void remove();
  };

interface ChildElementToParentElement : ROElement{  /* Port #93 */
  };

interface GeneralElement : ROParentElement,
ParentElementToChildElement, ChildElementToParentElement, ROElement {
  };
```

TABLE OF CONTENTS

INDEX