Object-Oriented Modeling in Metaview

Yong Zhuang, Piotr Findeisen, Paul Sorenson
Department of Computing Science
University of Alberta
yongz@bnr.ca, find@cs.ualberta.ca, sorenson@cs.ualberta.ca

May 7, 1996

Abstract

Metaview is a metasystem that can generate automatically environments to support software engineering activities, such as requirement analysis and design. This report focuses on creating Object-Oriented (O-O) support environments in Metaview to help efficiently construct reliable and maintainable software systems. Metaview's capability to define O-O CASE environments is described and critiqued. One innovative approach supported in Metaview is the use of aggregation, as a mechanism for linking method fragments. This approach is illustrated in the specification of our environment to support Object Modeling Technique (OMT), one of the most popular O-O development methods. OMT has been prototyped in Metaview and the results of this activity are analyzed.

1 Introduction

The dramatically increasing complexity of many software systems has led to our current "crisis" in software development. *Computer-Aided Software Engineering* (CASE) has evolved to help solve this problem by providing automated assistance for development teams to manage this complexity. An integrated CASE *environment* provides automated or semiautomated support for the system development that is guided by a corresponding method. Because it is costly to produce efficient and reliable support environments, meta system facilities, such as *Metaview* [McA88, STM88, DeD91, STM91], *MetaPlex* [CN89], *MetaEdit* [Smo91] and *Socrates* [Ver91], are being developed to generate automatically a variety of software specification environments to support major parts of CASE activities, such as requirement analysis and design. Each metasystem has an underlying model framework to develop software specification environments that are defined formally and unambiguously. This approach also reduces the time and cost of developing a specification environment significantly.

1.1 Motivation

The traditional structured methods of software development suffer from problems [KM90] that include little or no iteration within development phases, no emphasis on reusability and extendibility issues, and no unifying models and notation to integrate well the development phases. The 1980s brought a major breakthrough in software engineering with the popularization of object-oriented (O-O) approaches to system development [HS91]. These approaches model the real world at many levels of abstraction and thereby support the development of more comprehensible, reusable, and extendable software systems. In addition, O-O methods hold the promise of improving both the quality and the productivity of software development.

Prior to this work, several specification environments were modeled using Metaview [ST93]; however, an O-O environment had yet to be implemented. Although a number of O-O methods have been proposed, no standard has emerged and each existing O-O method suffers from some disadvantages.

Among several candidate object-oriented methods, Rumbaugh *et al*'s OMT [Rum91, Rum94, Gil94] was the first one chosen for modeling primarily because it has a relatively complex and expressive model, and it is one of the most popular methods among those being adopted by the industry.

Prototyping OMT as a representative O-O method using Metaview allows for some initial exploration of the complications of modeling O-O methods and thereby provides valuable insights into how we can improve our metasystem approach. In addition, on the basis of OMT modeling experience, we are better able to compare and critique popular O-O methods, propose object-oriented method enhancements, and implement them in Metaview. It is also hoped that much of the OMT definition can be reused for any enhanced method developed. Our explorations in enhancing O-O methods are beyond the scope of this paper, but some initial results are reported in [Zhu94].

1.2 Outline of the Paper

The rest of this paper is organized as follows. Section 2 introduces Metaview and presents its most important features. The OMT method is briefly described in section 3. Section 4 presents the modeling of OMT in Metaview. The use of aggregation to link method fragments is discussed in section 5. Section 6 concludes the paper with an analysis of Metaview's strengths and weaknesses, and a discussion of future research. Finally, Appendix **??** presents the complete EDL code for OMT as actually used within the Metaview system.

2 An Overview of Metaview

This section introduces Metaview's fundamental concepts and architecture [STM91, Fin92a, Fin92b, Fin93]. The modeling schema for software methods is also discussed. Metaview has three primary advantages not always present in other metasystems:

- *integrating a new environment*
  Metaview has mechanisms to integrate and maintain a variety of environment models; a newly defined model can be composed with other Metaview created models rather easily.

- *defining constraints*
  Metaview has a strong capability for defining system constraints during the environment configuration, which makes it possible to check the system consistency and completeness automatically.

- *evolving environments*
  An environment definer can easily alter or extend an existing environment to generally improve a specification environment or to match the software development requirements for a particular project. In addition, because of the common modeling framework, it is easy to compare alternative environments [ST93].

2.1 Fundamental Concepts of Meta-CASE systems

In the discipline of software engineering several methods of describing software specification are used. Many of them present the specification graphically in the form of diagrams. There is a variety of different describing techniques, which are applicable at various stages of software development. We will call these techniques *Software Description Models* (SDM). It should be emphasized that SDMs can be used not only for software development, but for specifying a given artifact as well. Most popular software description models include Data Flow Diagrams or Structure Charts, used in Structured Analysis and Design, and a plethora of object-oriented methods.

In a traditional CASE tool, the SDM (or a few SDMs) is given *a priori*. In contrast, a meta-CASE system is open for an unlimited number of SDMs. This feature can be achieved by storing not only the software specification, but also the SDM which was used to build it. The "language" used to describe SDMs, also called the *meta-model*, is specific for each meta-CASE system. The meta-model used in Metaview is EARA/GE, described in the next subsection.

Although most of the SDMs used in practice today feature graphical representation of the information (diagrams), it is important to distinguish between the *conceptual* and the *graphical* information. The graphical information contained in diagrams, such as size of various elements, length of lines and the general layout, directly affects diagram readability but is usually independent conceptually from the software artifacts represented in the diagram. The software repository must store both aspects, since users will prefer the graphical presentation, while most, if not all information about the software artifacts is stored from a conceptual perspective.

To be useful, a CASE tool must at least be able to check consistency and completeness of the software specification. Most of this activity is related to the conceptual aspects. In Metaview, the logical conditions to be satisfied by the specification are called *constraints*.

2.2 The EARA/GE meta-model

The meta-model defined for the Metaview system is called EARA/GE (Entity – Aggregate – Relationship – Attribute with Graphical Extension). As the name suggests, EARA is based on the entity-relationship data model, but incorporates significant extensions, like specialization, aggregation and some elements of generalization.

The basic concepts used in the EARA model are *entities*, *aggregates*, *relationships*, and *attributes*. Entities represent real-world objects. Typical entities used in SDMs are modules, procedures, documents, data, events, states, processes, variables, actions, data stores, etc. Relationships are associations that exist between entities and/or aggregates. For example, the fact that one module calls another can be expressed as a relationship between these modules. Similarly, we can use a relationship between a document and a module to indicate that the document describes the module. The entities and aggregates that take part in a relationship are called *participants* of that relationship. Relationships have *roles*. We assume that each relationship can be defined as a mapping from its roles to its participants. The roles determine the functions played by the participating entities.

To provide suitable support for complex objects, the model is capable of representing a heterogeneous collection of entities and relationships as an aggregate. The entities and relationships belonging to an aggregate are called *components* of that aggregate. Very often the entities defined by an SDM do not have a simple, atomic structure, but rather represent complex subsystems. The subsystems can be represented by aggregates. To give the capability of representing hierarchical systems, or presenting a software specification at the desired level of granularity, the EARA model uses the notion of *aggregation*. Aggregation is a special association between an entity and an aggregate. The participating aggregate is called *child aggregate* and the entity is called *parent entity*. The aggregate which the parent entity is a component of is called *parent aggregate*. Basically, aggregation means that the given entity represents the aggregate, or that the aggregate describes more details of the entity.

The child aggregate can optionally contain copies of some objects (entities and relationships) from the parent aggregate. These copies are called *boundary* objects. The boundary objects play a role similar to that of formal parameters in traditional programming. Mapping of the boundary objects to their counterparts within the parent aggregate is a part of the aggregation. Any specific requirements related to aggregation can be expressed by the constraints. When there are no constraints related to aggregation, and there are no boundary objects, the aggregation is similar to hypertext-like links, which allow the user to merely traverse the software specification, going from one aggregate to another.

Entities, relationships and aggregates can have *attributes*. The attributes represent properties of the objects and are additional means of refining software specifications.

The EARA model introduces classification of all entities, aggregates, relationships and attributes that can exist in a software specification by defining *object types*. Each entity, relationship or aggregate has a type. The objects of the same type share a lot of properties, like aggregation capabilities for entities, roles for relationships, eligible types of components for aggregates and the names and types of attributes. The object types can form *specialization hierarchies*. The hierarchy is formed when an object type is defined as a *subtype* of another type. Subtypes inherit all the properties of their supertypes.

The EARA model allows the uniform representation of both a particular software specification and the SDM that is used to build the specification. Thus there are two levels in the EARA repository: the *method level* is used to formally describe various SDMs and the *specification level* is used to define actual designs, or, in general, actual software specifications. The implementation of the EARA model includes a database storing both parts of the actual application.

The Graphical Extension (GE) adds the graphical aspects to the EARA model. The graphical counterparts for each of the conceptual types existing in the EARA model are defined. In particular, the graphical notation for *icon, edge* and *diagram types* are provided as the respective counterparts of *entity, relationship* and *aggregate types*.

To provide the capability of checking the consistency and completeness of software specifications, modeled SDMs are complemented by sets of *constraints*, or logical predicates, that must be satisfied by any "correct" software specification. There are also some meta-constraints, which are applicable to all specifications, independent of the SDM used. The constraints fall into one of the two categories:

- *consistency constraints*, which guard the consistency of the specification database, i.e. check if the specification "makes sense",
- *completeness constraints*, which ensure that the software specification is complete, i.e. check if there's no information missing.

The constraints can be related to both conceptual and graphical layer of the software specification.

Modeled SDMs can be described in a specially designed language called Environment Definition Language (EDL) (see [McA88] or [GLM94]). The features of the language strictly correspond to the concepts contained in EARA/GE. EDL will be used throughout section 4 in examples illustrating the modeling of OMT.

3 Object Modeling Technique

This section sketches the Object Modeling Technique (OMT), introduced by [Rum91]. The lack of space prevents us from presenting it in its entirety, and therefore only the most important features will be covered. Furthermore, some of the OMT details will be included in section 4, when discussion of the actual modeling takes place.

OMT proposes to develop software systems in four stages: analysis, system design, object design and implementation. All of these stages use to some extent the same SDMs: *Object Model*, *Dynamic Model* and *Functional Model*. In this paper we entirely ignore the development stages, or software development process, and focus on the product, or structural aspect of the software development.

The Object Model describes the structure of object classes, various relationships between them, and their attributes and operations. Graphically, the classes are represented by boxes, and the relationships by arcs (lines) linking the appropriate boxes. Binary and n-ary relationships exist. The roles of relationships can be optionally named, or given multiplicity. Several special ("built-in") relationships exist, like specialization, or aggregation[1]. Their graphical representation is different from the regular relationships and includes ornaments drawn on the arcs.

The Dynamic Model shows the state-transition diagrams for the classes introduced in the Object Model. A state-transition diagram contains different states in which an object of the given class may be, and transitions between the states. The transitions are triggered by events. The events correspond to operations defined within the Object Model. The transitions may optionally contain a condition and action. Graphically, the states are represented by ellipses, and transitions by arrows between them. The events, conditions and actions are placed along the arrows. The state-transition diagrams can be nested, to show multi-level activity.

The Functional Model specifies data transformation within the system. It uses a slightly modified version of data flow diagrams. The diagrams show processes, actors and data stores as nodes, and data flows as the arcs between them. Processes can be expanded into entire data flow diagrams.

4 OMT Environment Modeling

4.1 The Object Model

The conceptual contents of Object Model diagrams can be presented by the aggregate type `object_model`. To simplify the modeling through the use of subtyping, an abstract entity type called `om_entity` (object model entity) and an abstract relationship type called `om_relationship` are defined as the component types of the aggregate. These types will be supertypes for all entity and relationship types used within the Object Model.

```
AGGREGATE_TYPE object_model
   COMPONENTS (om_entity, om_relationship);
ENTITY_TYPE om_entity GENERIC;
RELATIONSHIP_TYPE om_relationship GENERIC;
```

[1]These concept in OMT are different from the analogous concepts in Metaview.

Figure 1 shows an actual Object Model diagram drawn by Metaview. We will use it to illustrate the constructions described in this section.

Figure 1: A sample Object Model diagram drawn by Metaview

Central notions in OMT's Object Model are class and association (between classes). The classes can be represented by EARA entities. However, it is impossible for the associations to be represented by EARA relationships. This is because the associations in Object Model can participate in other associations, while the EARA relationships can take only entities as participants.

To reflect all dependencies from the Object Model, we have to map associations to entities. Thus, each binary association binding two classes in the Object Model will be represented by an entity representing the association itself (type `binary_association`), and two relationships (of type `binding`) which bind the association entity to the corresponding class entities (type `class`). Another advantage of this approach is that it allows us to easily render the special shapes (diamond, triangle) that should appear on some of the edges. The shapes are supported by the entities representing the associations (see Figure 4 later in this section).

One of the more complicated constructions in the Object Model is the qualified association. It involves two classes, an association, and a qualifier. The qualifier is a special attribute that is written in a small box (LAN address in Figure 1) at one end of the association line and adjacent to one of the classes.

The qualifier can be represented by an entity, but its connections with the class and the association must be also modeled at the conceptual level. We do this by introducing the relationship type `qualification` that binds the qualifier and the class. Thus, the elements needed to build a qualified association have the following EDL definition:

```
ENTITY_TYPE any_class IS_A om_entity;
ENTITY_TYPE qualifier IS_A om_entity;
RELATIONSHIP_TYPE qualification IS_A om_relationship
   ROLES (class, qualifier)
   PARTICIPANTS (any_class, qualifier);
```

```
ENTITY_TYPE association GENERIC IS_A om_entity
    ATTRIBUTES (assoc_name : string);
ENTITY_TYPE binary_association IS_A association;
ENTITY_TYPE role IS_A om_entity
   ATTRIBUTES (role_name : identifier);
```

The n-ary associations (Connects in Figure 1), OMT's aggregation (like between Keyboard and Workstation in the figure) and specialization (like between Computer and Workstation) are modeled in a way very similar to that of binary associations, using the entity types n_ary_association, part_of and is_a, respectively. The complete hierarchy of entity types used for Object Model is presented in Figure 2.

Figure 2: The entity types used in Object Model

The binding relationship type has three subtypes: binds (like between Connects and Workstation in Figure 1), binds_many and binds_optional, which reflect the cardinality of the association.

A similar taxonomy is applied to the relationship types aggregation_from and aggregation_to, which together with the entity type is_a, are used to model OMT's aggregation. For example, aggregation_from_many is used for Disk Drive, and aggregation_from_optional is used for Mouse. Additional information about cardinality can be provided by the multiplicity attribute (displayed as the label next to CPU in the figure). For brevity, we show the EDL definitions for the hierarchy of bindings only. Other relationship types are defined in a very similar way.

```
RELATIONSHIP_TYPE binding GENERIC IS_A om_relationship
   ROLES (association, role, participant)
   PARTICIPANTS
      (binary_association, role, any_class | qualifier);
RELATIONSHIP_TYPE binds_optional IS_A binding;
RELATIONSHIP_TYPE binds IS_A binding
   PARTICIPANTS
      (n_ary_association, role, any_class);
RELATIONSHIP_TYPE binds_many IS_A binding
   ATTRIBUTES (multiplicity : string);
```

The three roles defined for the `binding` relationship type (`association`, `role`, and `participant`) are inherited by all its subtypes. Similarly, the set of participants defined for this relationship type is also inherited. The vertical bar represents alternative. Thus, the possible combinations of participants for `binding` include a `binary_association` entity for the `association` role, a `role` entity for `role`, and an `any_class` or a `qualifier` for the third, `participant` role. An additional combination of participants, which includes an `n_ary_association`, is defined for the `binds` relationship type only.

The complete hierarchy of relationship types for Object Model is presented in Figure 3.

Figure 3: The relationship types used in Object Model

In the Object Model, as originally proposed in [Rum91], the names of attributes and operations of a class are written inside the box representing the class. Again, this dependency must be somehow reflected at the conceptual level. One way of doing this might be introducing a new relationship type which would bind an attribute (or operation) to the class. However, the current version of the Graphical Extension for Metaview is not able to reliably place the operations and attributes within the class icon. Besides, Metaview has a mechanism that seems to be more suitable to represent attributes and operations of a class: aggregation.

To use aggregation for representing attributes and operations of classes, we introduce a new aggregate type `class_contents`. We also define two entity types: `attribute` and `operation` to represent the corresponding artifacts of the class. Finally, we declare `class_contents` as a child aggregate type of the `class` entity type.

```
AGGREGATE_TYPE class_contents
   COMPONENTS (attribute, operation);
ENTITY_TYPE attribute;
ENTITY_TYPE operation;
```

The aggregation is also used to link class entities to their corresponding state-transition diagrams (Dynamic Model). Additionally, Metaview allows the user to build a hypertext-like aggregation link from an entity to an external file. We use this capability here to link the file containing the source code of a class to the class icon[2]. Thus, we rewrite the definition of the class entity type as:

```
ENTITY_TYPE any_class GENERIC IS_A om_entity
   BECOMES class_contents, dynamic_model, source_code;
AGGREGATE_TYPE source_code IS_A file;
```

To be represented on the computer screen, all the conceptual types must be complemented by the graphical definitions. The Graphical Extension simplifies this job by allowing the user to define a hierarchy of the graphical types. For example, if `binds_many` and `aggregate_from_many` have the same graphical representation, it is sufficient to define it only once, to be shared by both relationship types. Figure 4 shows the graphical representations for some of the conceptual types defined for the Object Model. The names of attributes to be displayed as labels are shown in *italics*.

Figure 4: Graphical modeling for Object Model

Please note, that some of these types (e.g. `aggregate_to` and `qualification`) are not part of the OMT method as we know it. In fact, these elements usually do not show up in Metaview diagrams either. It is, however, convenient for the Metaview users when all conceptual artifacts are reflected in the graphical domain, too. For example, in Figure 1 `aggregation_to` is used to link Workstation with the icon of the `part_of` type. If that relationship was invisible, the user might move the `part_of` icon away (and even put it below another class icon), obtaining a mismatch between the conceptual and the graphical information. Graphical completeness constraints, which define what a "good looking" diagram is, can aid the user in "hiding" the needless edges.

[2]This capability exceeds the OMT's expressiveness as described in [Rum91].

4.2 The Dynamic Model

To model the dynamic behaviour of a software system, we use only one aggregate type, `dynamic_model`. We define top level entity and relationship types for the Dynamic Model as follows:

```
AGGREGATE_TYPE dynamic_model
   COMPONENTS (dm_entity, dm_relationship);
ENTITY_TYPE dm_entity GENERIC;
RELATIONSHIP_TYPE dm_relationship GENERIC;
```

There are two kind of states: (regular) states and final states. Their graphical representation is different, and no transitions are allowed from the final states. Entire state diagrams can be nested within the states. We model this by aggregation:

```
ENTITY_TYPE any_state GENERIC IS_A dm_entity
   ATTRIBUTE (state_name : string);
ENTITY_TYPE state IS_A any_state
   BECOMES dynamic_model
   ATTRIBUTES (activity : string);
ENTITY_TYPE final_state IS_A any_state;
```

The transitions are always triggered by events. We model those as entities, while the transitions are modeled by relationships. The optional conditions and actions are attributes of the relationship. They are displayed as labels along the edge representing the relationship.

The initial state is modeled as a one-role relationship. More than one initial state can be present within a diagram, if they correspond to different operations or are bound with different conditions. Figure 5 presents the graphical representation of the elements used in the Dynamic Model.

```
ENTITY_TYPE event IS_A dm_entity;
RELATIONSHIP_TYPE any_transition GENERIC IS_A dm_relationship
   ATTRIBUTES (condition : string);
RELATIONSHIP_TYPE transition IS_A any_transition
   ROLES (from, trigger, to)
   PARTICIPANTS (state, event, any_state)
   ATTRIBUTES (action : string);
RELATIONSHIP_TYPE initial_state IS_A any_transition
   ROLES (state)
   ATTRIBUTES (operation : string)
   PARTICIPANTS (any_state);
```

Figure 5: Graphical modeling for Dynamic Model

4.3 The Functional Model

Similarly as for the two previous model, we start with the definition of the aggregate type and the roots for the entity and relationship types hierarchy:

```
AGGREGATE_TYPE functional_model
   COMPONENTS (fm_entity, fm_relationship);
ENTITY_TYPE fm_entity GENERIC;
RELATIONSHIP_TYPE fm_relationship GENERIC;
```

The primary entities used in the Functional Model are processes, data stores and actors. Data flows between them are modeled by relationships. The `flow` relationships have three roles: one of them is occupied by an entity of the additional type `data`.

Processes can be expanded to show nested diagrams. Again, aggregation is used here. However, in this case we declare that all relationships in which a given process participates should be reflected in the child diagram as the boundary objects.

```
ENTITY_TYPE actor IS_A fm_entity;
ENTITY_TYPE process IS_A fm_entity
   BECOMES functional_model CONNECTIONS (flow);
ENTITY_TYPE data_store IS_A fm_entity;
ENTITY_TYPE data IS_A fm_entity;
RELATIONSHIP_TYPE flow IS_A fm_relationship
   ROLES (source, data, destination)
   PARTICIPANTS
      (process, data, actor | process | data_store)
      (actor | data_store, data, process);
```

The graphical modeling for the Functional Model is sketched in Figure 6.

The child diagram can contain multiple instances of boundary entities and/or relationships which are mapped onto a single instance in the parent diagram. Therefore, aggregation supports so called "data leveling", in which a single data flow in the parent diagram can be split into several flows within the child diagram.

Figure 7 shows an example of aggregation for data flow diagrams. The parent entity, packet dispatcher, is expanded into the child aggregate. In this simple case all objects in the

Figure 6: Graphical symbols used in the Functional Model

Figure 7: Leveling in the Functional Model

child aggregate are boundary objects. Processes message reassembly and socket manager are mapped onto the parent entity, while i/o module is mapped onto the correspondingly named process within the parent diagram. The data elements network packet and packet header are both mapped onto the network packet within the parent diagram.

5 Linking the method fragments

5.1 Aggregation

Aggregation links are built explicitly by the user during the software development, in agreement with the SDM definition. They can be used not only for binding together two aggregates, but also to navigate the software specification. Figure 8 presents the summary of the aggregation use in the OMT modeling:

Figure 8: Aggregations in OMT

- A `class` entity from the `object_model` can be expanded into a `class_contents` aggregate, showing the attributes and operations defined for the class.
- A `class` entity from the `object_model` can be expanded into a `dynamic_model` aggregate, showing the behaviour of objects of the given class using state-transition diagrams.
- A `state` entity from the `dynamic_model` can be expanded into another `dynamic_model`, supporting nesting of the state-transition diagrams.

- A `class` entity from the `object_model` can be expanded into a file containing the source code for the class.
- A `process` entity from the `functional_model` can be expanded into another `functional_model` aggregate, supporting data flow diagrams leveling.

Metaview supports *multiple* aggregation. Figure 8 shows that there are up to three aggregation links possible for each of the `class` entities. Multiple aggregation allows an entity to be presented in several different views.

Some of the aggregation links are automatically checked by Metaview for consistency and completeness. Such is the case of expanding processes into new aggregates. All flows related to the process (incoming or outcoming) must be reflected by at least one matching flow in the child aggregate.

On the other hand, some of the aggregation links are weak, like those between classes and their source files. While in the current version of Metaview such links cannot be supported by the boundary objects or constraints (Metaview sees an external file as a "black box"), future enhancement to the system may include necessary tools (e.g., parser), which will take full advantage of this form of aggregation.

5.2 Constraints

A constraint falls into one of either the *consistency* or *completeness* categories. The consistency constraints are checked whenever the software specification is modified by the user. If any is violated, the change is rejected. In contrast, the completeness constraints are checked on the user's demand, presumably when the user assumes that his/her work is complete. Their violation does not entail any consequences, since it is assumed that the user will continue the development until all the constraints are satisfied.

Some of the constraints can be related to aggregation. For example, if there's a state-transition diagram associated with a class, we might require (for completeness) that each event within the diagram has a corresponding (i.e. similarly named) operation defined for the class. In an analogous way, the actions existing within the state-transition diagram, should be recognizable as operation names for some classes associated with the given class. Used this way, the constraints complement the aggregation mechanism and can be used to build sophisticated links between aggregates.

6 Conclusions

The OMT support environment has been successfully defined using Metaview. Based on the experience of prototyping the OMT Method, the advantages and disadvantages of the Metaview metasystem's modeling capability are presented. Finally, we will present the future research areas.

6.1 Metaview's major strengths

- *Object Types*
  Metaview's EARA model supports quite well classification of objects through its object types. As we found out, an O-O method can incorporate a relatively large number of entity and relationship types. Many of these types share common characteristics. *Specialization* makes O-O modeling efficient because subtyping can be used to define those characteristics only once for the appropriate supertypes.

- *Aggregation*
  The aggregation mechanism provides the means for leveling the software specification and for partitioning it into manageable pieces. When used in combination with different aggregate types, it provides support for multiple views. Version control, albeit unimplemented in the current version of Metaview, can be also based on aggregation ([McA88]).

- *Constraints*
  One of the most important strong points of the Metaview system is the use of the constraints. Many diagramming techniques are imprecise. Modeling them in Metaview enforces the method definer to define clearly which constructions are allowed, thus providing a form of the method's "semantics". Since the constraints are based on first-order predicate logic, there are virtually no restrictions as to what can be expressed by the constraints. The constraints are particularly useful for relating together information contained in different method fragments (across-model constraints). This capability is not commonly found in current CASE tools such as Cadre's Teamwork [Inc93].

- *Reuse*
  SDMs defined in EDL can be reused. Two of the OMT's models: the Functional Model and the Dynamic Model are not new. They have been published and used independently for many years. Since we had gone through the exercise of modeling them previously, our current effort was mainly spent on incorporating minor changes to their definitions. These changes were partially due to the specific flavour for these method fragments as defined for OMT, and partially to the need of linking those fragments together. In fact, most of the modeling effort goes into the constraint definitions, but even those constraints which are entirely "embedded" in a particular method fragment, can be reused.

- *Common Transparent Repository*
  Metaview provides a schema to store all the analysis and design results produced in support environments that are prototyped using Metaview. The downloading of files and the access mechanisms of the repository are transparent to the analyst/developer.

- *Experimental Methods*
  The meta-system approach allows the method designer to concentrate on the logical aspects of the method. The appropriate support environments can be built from the

EDL definitions without any programming effort. This makes it easy to experiment with the methods, refine them frequently and assess their suitability for a particular task.

6.2 Metaview's Limitations

Generally, the main source of the limitations of the system is the Graphical Extension (GE). Some of the limitations stem from the current implementation of GE, but some are also inherently buried in its design.

- *Nested Elements*
  A major difficulty in modeling an environment is the graphical representation of nested diagrams, or icons. The aggregation provides an alternative solution, but it is not always a desired one.

- *Aggregation*
  Currently there's no way to graphically represent the aggregation, since each aggregate is displayed in an individual (top-level) window. While the navigation through the aggregates is well supported, the user can get confused when many diagrams are simultaneously open on the screen. The problem is especially visible when a form of aggregation with rich boundary information is used. The mapping from the boundary objects to the objects in the parent aggregate cannot be represented graphically.

- *Graphical Editor*
  The graphical editor used by Metaview for both introducing the specification and displaying it on the screen, is a simple tool without all the "bells and whistles" which often accompany commercial products. In particular, in the current version there's no support for grouping objects on the screen (selecting, moving, copying, collapsing to an aggregate, or deleting groups of objects).

- *Conditional Graphical Properties*
  Very often an environment uses very similar graphical representations for very similar conceptual objects, like within the `binding` hierarchy introduced in section 4. The only difference between the edges representing various relationships from the hierarchy is the edge terminator. The original design of GE provides the means for defining the graphical elements (like the terminators) based on the relationship's attribute value, for example. Full implementation of GE would reduce the graphical hierarchy for `binding` to just one element, with a conditional definition for the terminator. Both the method definer and the final user would benefit from this simplification.

- *More Flexibility for Specifications*
  Currently, it is impossible in Metaview to create an instance of a relationship when not all of the prospective participants exists. This was not seen as a major limitation during the design of Metaview, since eventually all participants have to be defined anyway. But this feature is not user-friendly, as we have learned. In particular, deleting a

participant entails deleting the relationship in which it participates. Removing this limitation does not require modifications of EARA and should be relatively easy to achieve.

- *Approach Too Formal?*
  One of the goals for the Metaview design was to provide a formal background for both the method and the software specification. As a result, while modeling a method which lacks the necessary formalism, the modeler is forced to provide it. While it can be argued that such a modeled method is "better" than the original one, the users of the system may have a different opinion.

  During the modeling of OMT, we encountered a number of questions or problems which were not adequately addressed in the book [Rum91]. We solved them either by analysing the enclosed examples of diagrams, or using our own judgement.

6.3 Suggestions of Future Research

There are several areas of future research and prototype development:

- *Integrating with other CASE tools*
  In the modeling of OMT we used aggregation as a "dumb" link from a class entity to an external file. We plan to investigate the possibility of enhancing this type of link not only to give it the full benefits of aggregation, but also to provide a mechanism for integration with other CASE tools.

- *Measuring the complexity of Models*
  As first noted by Rossi and Brinkkemper [RB95], a major advantage of using a meta model as a basis for developing a specification environment is that key size measures related to the complexity of an environment can be easily derived (e.g., number of entity, relationship, attribute and aggregate types along with various statistics related to constraint definitions). We have not reported these here because further work is required to model fully the other O-O environments. Also, in the process of modeling OMT we monitored the amount of effort required to create and modify model definitions. The complexity results of this work will be reported in the near future.

- *Improving Metaview's Facilities*
  As mentioned in section 6.1, Metaview's modeling limitations, such as the EDL/ECL's expression capability, or graphical editing function, should be improved.

- *Adding Enactment Engine*
  Currently Metaview supports only static software specifications, i.e. diagrams. There's no support for automatic or semi-automatic transformations between software specifications, and no support for interpreting the specifications (in cases when the specifications are executable). Although software development processes can be modeled in Metaview, such an activity would be rather pointless without any underlying execution

engine. We plan to continue our work in this direction, and to integrate our former partial results into Metaview. The Enactment Engine will be ECA-rules (event - condition - action) based [FTS95] and will support software process modeling and enactment, transformations between software specifications and specifications execution.

- *Supporting Integrated Methodologies and Integrated Environments*
  Previous work have been done to support multiple views for an integrated specification environment [Zhu93] and to support transformation processes between different structured methods [Lee92]. The challenge exists of integrating these research efforts to explore multiple-view environments that support process transformations between different O-O methods.

6.4 Summary

This paper has explored the modeling and implementation of an OMT specification environment using a metasystem tool Metaview. The major results from this work are:

i) Through our prototype we discovered that we can quickly develop, change, reuse and evolve O-O models like OMT using a metasystem approach, and

ii) Links between method fragments in a method can be enhanced through the use of multiple aggregation and constraint rules. Those aspects, when combined, provide a powerful and unique capability for Metaview.

6.5 Metaview Implementation

The Metaview system is under continuous development at the Computing Science Department, University of Alberta. The current prototype works on Unix platforms (Sun, IBM). It has been implemented using Prolog and C++.

From the CASE user perspective, the system provides Motif-based graphical user interface and multi-access to the software database. At present, the EDL/GE implementation does not support the constraints, which have to be “manually” encoded in Prolog. An enhanced version of ECL, our constraint language, is currently being developed.

More information about the Metaview project is available on the Internet’s World Wide Web at `http://web.cs.ualberta.ca/~softeng/Metaview/project.html`. Extensive documentation of the system is also available there.

References

[CN89] M. Chen and J. F. Nunamaker. MetaPlex: an integrated environment for organization and information System Development. *Proceeding of Tenth International Conference on Information System*, pages 141–151, 1989.

[DeD91] J. M. DeDourek, P.G. Sorenson, and J.P. Tremblay. Metasystems for Information Processing System Specification Environments. *INFOR*, 27(3):331–337, August 1991.

[Fin92a] Piotr Findeisen. The EARA/GE Model for Metaview. Department of Computing Science, University of Alberta[1], November 1992.

[Fin92b] Piotr Findeisen. The Graphical Extension for EARA Model. Department of Computing Science, University of Alberta[1], September 1992.

[Fin93] Piotr Findeisen. The Metaview software. Department of Computing Science, University of Alberta[1], March 1993.

[FTS95] Garry Froehlich, J.Paul Tremblay, and Paul Sorenson. Providing support for process model enaction in the Metaview system. *Proceeding of the Seventh International Workshop on Computer-Aided Software Engineering*, pages 141–151, 1995.

[Gil94] C. Gilliam. An Approach for Using OMT in the Development of Large Systems. *Journal of Object-Oriented Programming*, pages 56–59, February 1994.

[GLM94] Dinesh Gadwal, Pius Lo, and Beth Millar. EDL User's Manual. Department of Computing Science, University of Alberta[1], June 1994.

[HS91] Brain Henderson-Sellers. *A Book of Object-Oriented Knowledge: Object-Oriented Analysis, Design and Implementation: a New Approach to Software Engineering.* Prentice Hall, 1991.

[Inc93] Cadre Technologies Inc. *Teamwork/OOA User's Guide.* 222 Richmond St. Providence, RI 02903-9990, release 5.0 edition, 1993.

[KM90] Tim Korson and John D. McGregor. Understanding Object-Oriented: a Unifying Paradigm. *Communications Of The ACM*, 33(9):40–60, September 1990.

[Lee92] Jesse Ka-Leung Lee. Implementing ADISSA transformations in the Metaview meta-system. Master's thesis, Department of Computing Science, University of Alberta, 1992.

[McA88] Andrew J. McAllister. *Modeling Concepts for Specification Environments.* PhD thesis, Department of Computational Science, University of Saskatchewan, 1988.

[RB95] M. Rossi and S. Brinkkemper. Metrics in method engineering. *Proceeding of the Seventh International Conference on Advanced Information Systems Engineering (CAiSE'95)*, pages 200–216, 1995.

[Rum91] James Rumbaugh, et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[1]Also available at `http://web.cs.ualberta.ca/~softeng/Metaview/system/documentation.html`

[Rum94] James Rumbaugh. The Life of an Object Model: How the Object Model Changes During Development. *Journal of Object-Oriented Programming*, pages 24–32, March 1994.

[Smo91] K. Smolander, K. Lyytinen, V. P. Tahvanainen, and P. Martin. MetaEdit: A Flexible Graphical Environment for Methodology Modeling. *Advanced Information System Engineering: Third Intl'l Conf. of CAiSE'91*, pages 168–191, May 1991.

[ST93] P.G. Sorenson and J.P. Tremblay. Using a Metasystem Approach to Support and Study the Design Process. *Workshop on Studies of Software Design*, pages 168–183, May 1993.

[STM88] Paul G. Sorenson, Jean-Paul Tremblay, and Andrew J. McAllister. The Metaview System for Many Specification Environments. *IEEE Software*, 5(2):30–38, March 1988.

[STM91] Paul G. Sorenson, J. Paul Tremblay, and Andrew J. McAllister. The EARA/GI Model for Software Specification Environments. Technical Report TR 91-14, University of Alberta, June 1991.

[Ver91] T. F. Verhoef, A. ter Hofstede, and G. M. Wijers. Structuring Modeling Knowledge for CASE Shells. *Advanced Information System Engineering: Third Intl'l Conf. of CAiSE'91*, pages 502–524, May 1991.

[Zhu93] Yuchen Zhu. Multiple views for integrated CASE environments. Master's thesis, Department of Computing Science, University of Alberta, 1993.

[Zhu94] Yong Zhuang. Object-oriented modeling in Metaview. Master's thesis, Department of Computing Science, University of Alberta, Fall 1994. Also available at `http://web.cs.ualberta.ca/~softeng/Theses/zhuang.html`.

A The EDL code

This appendix presents the EDL code describing OMT. The code contains complete conceptual and graphical definitions. It was compiled by an EDL compiler and the resulting object code was then used to configure the Metaview system. The configured system was tested for a few examples. In particular, Figure 1 on page 7 has been drawn using Metaview and the code shown below.

```
ENVIRONMENT_TITLE "Object Modeling Technique";

/****************************************************************
 *                                                              *
 *    Rumbaugh's Object Modeling Technique.                     *
 *    Modeled for Metaview by Y. Zhuang and P. Findeisen        *
 *                         University of Alberta 1994, 1995       *
 *                                                                 *
 *    The OMT consists of three models:                            *
 *       - object model,                                           *
 *       - dynamic model,                                          *
 *       - functional model.                                       *
 *    These three models have been described as independently as   *
 *    possible in the following EDL/GE text. Additionally, concep- *
 *    tual and graphical element are clearly separated.            *
 *                                                                 *
 ******************************************************************/

/******************************************************************
 *                                                                 *
 *              Object Model - Conceptual Definitions              *
 *                                                                 *
 ******************************************************************/

AGGREGATE_TYPE object_model
    COMPONENTS
       (om_entity, om_relationship);

ENTITY_TYPE om_entity GENERIC;
RELATIONSHIP_TYPE om_relationship GENERIC;

ENTITY_TYPE any_class GENERIC IS_A om_entity
    BECOMES dynamic_model, class_contents;

ENTITY_TYPE class IS_A any_class;
ENTITY_TYPE derived_class IS_A any_class;

/*  Associations are modeled as entities, primarily because they can
    participate in other relationships (association as class).    */

ENTITY_TYPE association GENERIC IS_A om_entity
    ATTRIBUTES (assoc_name : string);
ENTITY_TYPE binary_association IS_A association;
ENTITY_TYPE n_ary_association IS_A association;

ENTITY_TYPE role IS_A om_entity
        ATTRIBUTES (role_name : identifier);

ENTITY_TYPE qualifier IS_A om_entity;
ENTITY_TYPE is_a IS_A om_entity;
ENTITY_TYPE part_of IS_A om_entity;

RELATIONSHIP_TYPE qualification IS_A om_relationship
```

```
        ROLES (class, qualifier)
        PARTICIPANTS
           (any_class, qualifier);

RELATIONSHIP_TYPE binding GENERIC IS_A om_relationship
        ROLES (association, role, participant)
        PARTICIPANTS
           (binary_association, role, any_class | qualifier)
           (n_ary_association, role, any_class);

RELATIONSHIP_TYPE binds_optional IS_A binding;

RELATIONSHIP_TYPE binds IS_A binding;

RELATIONSHIP_TYPE binds_many IS_A binding
        ATTRIBUTES (multiplicity : string);

RELATIONSHIP_TYPE association_as_class IS_A om_relationship
        ROLES (association, class)
        PARTICIPANTS
           (association, class);

RELATIONSHIP_TYPE specialize_from IS_A om_relationship
        ROLES (superclass, structure)
        PARTICIPANTS
           (any_class, is_a)
        ATTRIBUTES (discriminator : string);

RELATIONSHIP_TYPE specialize_to IS_A om_relationship
        ROLES (structure, subclass)
        PARTICIPANTS
           (is_a, class);

RELATIONSHIP_TYPE aggregation_from GENERIC IS_A om_relationship
        ROLES (component, role, structure)
        PARTICIPANTS
           (any_class, role, part_of);

RELATIONSHIP_TYPE aggregate_from_many IS_A aggregation_from
   ATTRIBUTES (multiplicity : string);

RELATIONSHIP_TYPE aggregate_from IS_A aggregation_from;

RELATIONSHIP_TYPE aggregate_from_optional IS_A aggregation_from;

RELATIONSHIP_TYPE aggregation_to GENERIC IS_A om_relationship
    ROLES (structure, composite)
    PARTICIPANTS
       (part_of, any_class);

RELATIONSHIP_TYPE aggregate_to_many IS_A aggregation_to
   ATTRIBUTES (multiplicity : string);

RELATIONSHIP_TYPE aggregate_to IS_A aggregation_to;

RELATIONSHIP_TYPE aggregate_to_optional IS_A aggregation_to;


AGGREGATE_TYPE class_contents
    COMPONENTS
       (class_attribute);

ENTITY_TYPE class_attribute GENERIC;
ENTITY_TYPE attribute IS_A class_attribute;
ENTITY_TYPE operation IS_A class_attribute;

/*******************************************************************
 *                                                                 *
 *              Object Model - Graphical Definitions               *
 *                                                                 *
 *******************************************************************/

CONSTANT FontH     =        13;  /* Standard font height */


ICON_TYPE class_attribute IS_A primitive;

DIAGRAM_TYPE class_contents
    PROPERTIES (x_size = 150, y_size = 240);


DIAGRAM_TYPE object_model
    PROPERTIES (x_size = 640, y_size = 360);

CONSTANT ClassH    = 3*FontH+2;  /* The height of the class icon */
CONSTANT ClassW    =        80;  /* Class icon width  */

PICTURE_TYPE class_pic
    BOX FROM (0, 0) TO (ClassW, ClassH);

ICON_TYPE any_class GENERIC
    PROPERTIES (x_size = ClassW+1, y_size = ClassH+1)
    LABELS
        (name AT (ClassW/2, ClassH/2)
             PROPERTIES (x_size = ClassW-1, y_size = ClassH-2))
    HANDLES
        (qualification.class AT ((ClassW, 0..ClassH), /* right edge */
                                 (0..ClassW, ClassH), /* bottom edge */
                                 (0, 0..ClassH),      /* left edge */
                                 (0..ClassW,0)),      /* top edge */
         association_as_class.class AT ((ClassW/2, -1)))
    PICTURES
        (class_pic);

ICON_TYPE class IS_A any_class;

CONSTANT DerBar    =        12;  /* Derived bar parameter */

PICTURE_TYPE derived_bar
    LINE FROM (0, DerBar) TO (DerBar, 0);

ICON_TYPE derived_class IS_A any_class
    PICTURES (derived_bar);

CONSTANT BinAscW   =        61;  /* binary association icon width */
CONSTANT BinAscH   = 3*FontH+2;  /* binary association icon height */

ICON_TYPE binary_association
    PICTURES (point_pic AT (BinAscW/2-2, BinAscH/2),
              point_pic AT (BinAscW/2+2, BinAscH/2))
    /* the pictures should get obscured by the label, if present */
    LABELS
        (assoc_name AT (BinAscW/2, BinAscH/2))
    HANDLES
        (binding.association
            AT ((BinAscW/2,-1),       /* above the top */
                (-1, BinAscH/2),      /* left of the left edge */
                (BinAscW/2, BinAscH), /* bottom edge middle */
                (BinAscW, BinAscH/2)))/* right edge middle */
    PROPERTIES (x_size = BinAscW, y_size = BinAscH);

CONSTANT NaryAscW  =        69;  /* n-ary association icon width */
CONSTANT NaryAscH  = 3*FontH+6;  /* n-ary association icon height */

PICTURE_TYPE n_ary_pic           /* a diamond-like shape */
    LINE FROM (NaryAscW/2, 0) TO (0, NaryAscH/2)
    LINE FROM (0, NaryAscH/2) TO (NaryAscW/2, NaryAscH)
    LINE FROM (NaryAscW/2, NaryAscH) TO (NaryAscW, NaryAscH/2)
    LINE FROM (NaryAscW, NaryAscH/2) TO (NaryAscW/2, 0);

ICON_TYPE n_ary_association
```

```
    PROPERTIES (x_size = NaryAscW, y_size = NaryAscH)
    HANDLES
        (binds.association
            AT ((NaryAscW/2,0),         /* top edge middle */
                (0, NaryAscH/2),        /* left edge middle */
                (NaryAscW/2, NaryAscH), /* bottom edge middle */
                (NaryAscW, NaryAscH/2)),/* right edge middle */
         association_as_class.association
            AT ((NaryAscW/2, NaryAscH-1))) /* above the bottom edge */
    LABELS (assoc_name AT (NaryAscW/2, NaryAscH/2)
        PROPERTIES (x_size = NaryAscW-1, y_size = NaryAscH-1))
    PICTURES (n_ary_pic);

CONSTANT QualW     =        65;  /* qualifier icon width */
CONSTANT QualH     = 2*FontH+2;  /* qualifier icon height */

PICTURE_TYPE qualifier_pic
    BOX FROM (0, 0) TO (QualW-1, QualH-1);

ICON_TYPE qualifier
    LABELS
        (name AT (QualW/2, QualH/2)
            PROPERTIES (x_size = QualW-2, y_size = QualH-2))
    PROPERTIES (x_size = QualW, y_size = QualH)
    HANDLES
        (qualification.qualifier
               AT ((0,QualH/2),  /* middle of the left edge */
                   (QualW/2,0))) /* middle of the top edge */
    PICTURES (qualifier_pic);

CONSTANT PrimW     =        76;  /* primitives width */
CONSTANT PrimH     =     FontH;  /* primitives height */

ICON_TYPE primitive GENERIC
    PROPERTIES (x_size = PrimW, y_size = PrimH)
    LABELS (name AT (PrimW/2, PrimH/2));

CONSTANT IsaW      =        25;  /* is_a icon width */
CONSTANT IsaH      =        17;  /* is_a icon height */

PICTURE_TYPE is_a_pic            /* a triangle */
    LINE FROM (IsaW/2, 0) TO (0, IsaH-1)
    LINE FROM (0, IsaH-1) TO (IsaW-1, IsaH-1)
    LINE FROM (IsaW-1, IsaH-1) TO (IsaW/2, 0);

ICON_TYPE is_a
    PROPERTIES (x_size = IsaW, y_size = IsaH)
    HANDLES
        (specialize_from.structure
              AT ((IsaW/2, 0)),          /* top edge middle */
         specialize_to.structure
              AT ((0..IsaW-1, IsaH-1)))  /* bottom edge */
    PICTURES (is_a_pic);

CONSTANT PartW     =        17;  /* part_of icon width */
CONSTANT PartH     =        17;  /* part_of icon height */

PICTURE_TYPE part_of_pic         /* a diamond-like shape */
    LINE FROM (PartW/2, 0) TO (0, PartH/2)
    LINE FROM (0, PartH/2) TO (PartW/2, PartH-1)
    LINE FROM (PartW/2, PartH-1) TO (PartW-1, PartH/2)
    LINE FROM (PartW-1, PartH/2) TO (PartW/2, 0);

ICON_TYPE part_of
    PROPERTIES (x_size = PartW, y_size = PartH)
    HANDLES
        (aggregation_from.structure
             AT ((PartW-1, PartH/2),  /* the right cusp */
                 (PartW/2, PartH-1)), /* the bottom cusp */
         aggregation_to.structure
             AT ((0, PartH/2),        /* the left cusp */
                 (PartW/2, 0)))       /* the top cusp */
    PICTURES (part_of_pic);

PICTURE_TYPE point_pic
    POINT;

CONSTANT RoleW     =        51;    /* Role icon width */
CONSTANT RoleH     =     FontH;    /* Role icon height */

ICON_TYPE role
    PICTURES (point_pic AT (RoleW/2, RoleH/2))
    /* should be obscured by the label, if present */
    LABELS
        (role_name AT (RoleW/2, RoleH/2))
    PROPERTIES (x_size = RoleW, y_size = RoleH);

CONSTANT SpecD     =        50;  /* specialize edge length */

EDGE_TYPE specialize_from
    NODES
       (superclass AT (0, -SpecD),
        structure AT (0, 0))
    LINKS
```

```
       (FROM superclass TO structure
           LABELS (discriminator AT (9, 4)
               PROPERTIES (x_adjust = "left")))
    PROPERTIES (no_stretch = "x");

EDGE_TYPE specialize_to
    NODES (structure AT (0, -SpecD), subclass AT (0, 0))
    LINKS (FROM structure TO subclass);

EDGE_TYPE qualification
    NODES (class AT (0, 0), qualifier AT (1, 0))
    LINKS (FROM class TO qualifier
               PROPERTIES (dashing = "(2,3)"));

CONSTANT SCircR    =         4;  /* small circle radius */
CONSTANT SBallR    =         4;  /* small ball radius */

PICTURE_TYPE small_circle        /* for optional associations */
    CIRCLE CENTER (SCircR+1, 0) RADIUS SCircR;

PICTURE_TYPE small_ball          /* for multiple associations */
    CIRCLE CENTER (SBallR+1, 0) RADIUS 4
    CIRCLE CENTER (SBallR+1, 0) RADIUS 3
    CIRCLE CENTER (SBallR+1, 0) RADIUS 2
    CIRCLE CENTER (SBallR+1, 0) RADIUS 1;

CONSTANT BindD     =       100;  /* binds edge family length */
CONSTANT RoleXl    =        33;  /* X location of role */
CONSTANT RoleYl    =         9;  /* Y location of role */

EDGE_TYPE binds_optional
    NODES (association AT (BindD, 0),
           role        AT (RoleXl, RoleYl),
           participant AT (0, 0))
    LINKS
       (FROM association TO participant
           PICTURES (small_circle AT participant)
           PROPERTIES (cutoff_2 = 9, no_stretch = "y"));

EDGE_TYPE binds
    NODES (association AT (BindD, 0),
           role        AT (RoleXl, RoleYl),
           participant AT (0, 0))
    LINKS
       (FROM association TO participant);

EDGE_TYPE binds_many
```

```
    NODES (association AT (BindD, 0),
           role        AT (RoleXl, RoleYl),
           participant AT (0, 0))
    LINKS
       (FROM association TO participant
           LABELS (multiplicity AT (19, -9))
           PICTURES (small_ball AT participant));

PICTURE_TYPE half_ellipsis
    ARC CENTER (0,0) RADIUSES 8, 20 START 180 SPAN 180;

EDGE_TYPE association_as_class
    NODES (association AT (0,0), class AT (0, 20))
    LINKS
       (FROM association TO class
           PICTURES (half_ellipsis PROPERTIES (rotate = "no"))
           PROPERTIES (cutoff_1 = 20, dashing = "(1,2)"));

EDGE_TYPE aggregate_from_many
    NODES (component AT (BindD, 0),
           role      AT (RoleXl, RoleYl),
           structure AT (0, 0))
    LINKS
       (FROM component TO structure
           LABELS (multiplicity AT (BindD-20, -9)
               PROPERTIES (x_adjust = "left"))
           PICTURES (small_ball AT component ROTATED 180));

EDGE_TYPE aggregate_from
    NODES (component AT (BindD, 0),
           role      AT (RoleXl, RoleYl),
           structure AT (0, 0))
    LINKS (FROM component TO structure);

EDGE_TYPE aggregate_from_optional
    NODES (component AT (BindD, 0),
           role      AT (RoleXl, RoleYl),
           structure AT (0, 0))
    LINKS
       (FROM component TO structure
           PICTURES (small_circle AT component ROTATED 180)
           PROPERTIES (cutoff_1 = 9));

CONSTANT AgrTo     =        10;  /* "aggregate to" basic length */

EDGE_TYPE aggregate_to_many
    NODES (structure AT (AgrTo, 0),
            composite AT (0, 0))
    LINKS
       (FROM structure TO composite
           LABELS (multiplicity AT (5, -9)
               PROPERTIES (x_adjust = "left"))
           PICTURES (small_ball)
           PROPERTIES (dashing = "(1,3)"));

EDGE_TYPE aggregate_to
    NODES (structure AT (1, 0),
           composite AT (0, 0))
    LINKS
       (FROM structure TO composite
           PROPERTIES (dashing = "(1,3)"));

EDGE_TYPE aggregate_to_optional
    NODES (structure AT (AgrTo, 0),
           composite AT (0, 0))
    LINKS
       (FROM structure TO composite
           PICTURES (small_circle)
           PROPERTIES (cutoff_2 = 9));

/*******************************************************************
 *                                                                 *
 *     Dynamic Model (State Diagrams) - Conceptual Definitions     *
 *                                                                 *
 *******************************************************************/

AGGREGATE_TYPE dynamic_model
    COMPONENTS (dm_entity, dm_relationship);

ENTITY_TYPE dm_entity GENERIC;
RELATIONSHIP_TYPE dm_relationship GENERIC;

ENTITY_TYPE any_state GENERIC IS_A dm_entity
    ATTRIBUTES (state_name : string);

ENTITY_TYPE state IS_A any_state
    ATTRIBUTES (activity : string);

ENTITY_TYPE final_state IS_A any_state;

ENTITY_TYPE event IS_A dm_entity;

RELATIONSHIP_TYPE any_transition GENERIC IS_A dm_relationship
    ATTRIBUTES (condition : string);
```

```
RELATIONSHIP_TYPE transition IS_A any_transition
    ROLES (from, trigger, to)
    PARTICIPANTS (state, event, any_state)
    ATTRIBUTES (action : string);

RELATIONSHIP_TYPE initial_state IS_A any_transition
    ROLES (state)
    ATTRIBUTES (operation : string)
    PARTICIPANTS (any_state);

/******************************************************************
 *                                                                *
 *     Dynamic Model (State Diagrams) - Graphical Definitions     *
 *                                                                *
 ******************************************************************/

CONSTANT StBallR   =       5;  /*  Initial/final state ball radius */
CONSTANT StCircR   =      11;  /*  Final state circle radius */

PICTURE_TYPE state_ball
    POINT
    CIRCLE RADIUS 1
    CIRCLE RADIUS 2
    CIRCLE RADIUS 3
    CIRCLE RADIUS 4
    CIRCLE RADIUS 5;

PICTURE_TYPE state_circle
    CIRCLE RADIUS StCircR;

ICON_TYPE final_state
    PROPERTIES (x_size = 2*StCircR+1, y_size = 2*StCircR+1)
    HANDLES (*.* AT ((StCircR, StCircR)))
    PICTURES (state_ball AT (StCircR, StCircR),
              state_circle AT (StCircR, StCircR));

CONSTANT StateH    = 4*FontH+2;  /*  Regular state icon height */
CONSTANT StateW    =  3*StArcR;  /*  State icon box part width */
CONSTANT StArcR    =  StateH/2;  /*  Regular state arc radius */

PICTURE_TYPE state_pic
    LINE FROM (StArcR, StateH) TO (StArcR+StateW-1, StateH)
    LINE FROM (StArcR, 0) TO (StArcR+StateW-1, 0)
    ARC CENTER (StArcR, StArcR) RADIUS StArcR START 90 SPAN 180
    ARC CENTER (StArcR+StateW, StArcR)
        RADIUS StArcR START 270 SPAN 180
    TEXT "do:" AT (3, FontH+2);

ICON_TYPE state
    PROPERTIES (x_size = 2*StArcR+StateW+1, y_size = StateH+1)
    LABELS (state_name AT (StArcR+StateW/2, FontH/2+1)
               PROPERTIES (x_size = StateW+14, y_size = FontH),
            activity AT (StArcR+StateW/2, FontH*5/2+1)
               PROPERTIES (x_size = StateW+18, y_size = 3*FontH))
    HANDLES (transition.* AT
               ((StArcR-10 .. StArcR+StateW+9, 0 .. 2*StArcR)))
    PICTURES (state_pic);

CONSTANT EventH    =   2*FontH;  /* Event icon height */
CONSTANT EventW    =   4*FontH;  /* Event icon height */

ICON_TYPE event
    PROPERTIES (x_size = EventW+1, y_size = EventH+1)
    LABELS (name AT (EventW/2, EventH/2));

PICTURE_TYPE arrowhead
    LINE TO (-15, -5)
    LINE TO (-15, 5);

CONSTANT TransLabOff = FontH+5; /* Transition labels offset */
CONSTANT TransitionL = 160;     /* Transition edge length */
CONSTANT IniStateL   = 70;      /* Initial state edge length */

EDGE_TYPE transition
    NODES (from AT (0,0),
           trigger AT (TransitionL/3, EventH/2+5),
           to AT (TransitionL, 0))
    LINKS (FROM from TO to
      LABELS (
        condition AT (FontH, -TransLabOff)
          PROPERTIES (x_adjust = "left"),
        action AT to + (-FontH, TransLabOff)
          PROPERTIES (x_adjust = "right"))
      PICTURES (arrowhead AT to));

EDGE_TYPE initial_state
    NODES (state AT (0, IniStateL))
    LINKS (FROM (0, 0) TO state
      LABELS (operation AT (5, StBallR+FontH)
                PROPERTIES (x_adjust = "left"))
      PICTURES (state_ball AT (0, 0),
                arrowhead AT state ROTATED 270));
```

```
DIAGRAM_TYPE dynamic_model
    PROPERTIES (x_size = 640, y_size = 480);

/******************************************************************
 *                                                                *
 *         Functional Model (Data Flow Diagrams)                  *
 *                              - Conceptual Definitions          *
 *                                                                *
 ******************************************************************/

AGGREGATE_TYPE functional_model
    COMPONENTS (fm_entity, fm_relationship);

ENTITY_TYPE fm_entity GENERIC;
RELATIONSHIP_TYPE fm_relationship GENERIC;

ENTITY_TYPE process IS_A fm_entity
    BECOMES functional_model
        CONNECTIONS (any_flow, results_in);

ENTITY_TYPE data_store IS_A fm_entity;

ENTITY_TYPE actor IS_A fm_entity;

ENTITY_TYPE point GENERIC IS_A fm_entity;
ENTITY_TYPE split_point IS_A point;
ENTITY_TYPE replication_point IS_A point;

ENTITY_TYPE data IS_A fm_entity;
ENTITY_TYPE control IS_A fm_entity;

RELATIONSHIP_TYPE any_flow GENERIC IS_A fm_relationship;
RELATIONSHIP_TYPE control_flow IS_A any_flow
    ROLES (source, control, destination)
    PARTICIPANTS (process, control, process | actor)
                 (actor, control, process);

RELATIONSHIP_TYPE any_data_flow GENERIC IS_A any_flow;
RELATIONSHIP_TYPE data_flow IS_A any_data_flow
    ROLES (source, data, destination)
    PARTICIPANTS
        (process, data, process | actor | data_store | point)
        (actor | data_store, data, process | point)
        (split_point, data, process | actor | data_store);

RELATIONSHIP_TYPE replicated_flow IS_A any_data_flow
    ROLES (source, destination)
    PARTICIPANTS
        (replication_point, process | actor | data_store | point);

RELATIONSHIP_TYPE results_in IS_A fm_relationship
    ROLES (creator, object)
    PARTICIPANTS (process | replication_point, actor | data_store);

/*****************************************************************
 *                                                               *
 *         Functional Model (Data Flow Diagrams)                 *
 *                              - Graphical Definitions          *
 *                                                               *
 *****************************************************************/

CONSTANT ProcessRh = ProcessRv*3/2;  /* horizontal radius */
CONSTANT ProcessRv = FontH*5/2+1;    /* vertical radius */

PICTURE_TYPE process_pic
    ARC CENTER (ProcessRh, ProcessRv)
        RADIUSES ProcessRh, ProcessRv START 0 SPAN 360;

ICON_TYPE process
    PICTURES (process_pic)
    LABELS (name AT (ProcessRh, ProcessRv)
        PROPERTIES (x_size = ProcessRh*3/2+6, y_size = 4*FontH))
    PROPERTIES (x_size = 2*ProcessRh+1, y_size = 2*ProcessRv+1);


CONSTANT DStoreH     =  3*FontH+4;  /* Data store icon height */
CONSTANT DStoreW     =  2*DStoreH;  /* Data store icon width */

PICTURE_TYPE data_store_pic
    LINE FROM (0,1) TO (DStoreW, 1) PROPERTIES (thickness = 2)
    LINE FROM (0,DStoreH) TO (DStoreW, DStoreH)
      PROPERTIES (thickness = 2);

ICON_TYPE data_store
    PICTURES (data_store_pic)
    LABELS (name AT (DStoreW/2, DStoreH/2+1)
        PROPERTIES (x_size = DStoreW, y_size = 3*FontH))
    PROPERTIES (x_size = DStoreW+1, y_size = DStoreH+1);


CONSTANT DataW        =   DataH*5/2;  /* data icon width */
CONSTANT DataH        =     2*FontH;  /* data icon height */

ICON_TYPE data
```

```
    LABELS (name AT (DataW/2, FontH)
                PROPERTIES (x_size = DataW-1, y_size = 2*FontH))
    PROPERTIES (x_size = DataW, y_size = DataH);

ICON_TYPE control
    LABELS (name AT (DataW/2, FontH)
                PROPERTIES (x_size = DataW-1, y_size = 2*FontH))
    PROPERTIES (x_size = DataW, y_size = DataH);

CONSTANT ActorH    =   4*FontH+2;   /* Actor icon height */
CONSTANT ActorW    =  ActorH*3/2;   /* Actor icon width */

PICTURE_TYPE actor_pic
    BOX FROM (0, 0) TO (ActorW, ActorH);

ICON_TYPE actor
    PICTURES (actor_pic)
    LABELS (name AT (ActorW/2, ActorH/2)
        PROPERTIES (x_size = ActorW-2, y_size = ActorH-2))
    PROPERTIES (x_size = ActorW+1, y_size = ActorH+1);


PICTURE_TYPE dot_pic
    POINT
    CIRCLE RADIUS 1
    CIRCLE RADIUS 2
    CIRCLE RADIUS 3;

CONSTANT PointR    =  3;       /* Point icons radius */

ICON_TYPE point
    PICTURES (dot_pic AT (PointR, PointR))
    HANDLES (*.* AT ((PointR, PointR)))
    PROPERTIES (x_size = 2*PointR+1, y_size = 2*PointR+1);

CONSTANT FlowL    = 200;      /* Flow edge length */

EDGE_TYPE data_flow
    NODES (source      AT (0, 0),
           data        AT (FlowL/2, 20),
           destination AT (FlowL, 0))
    LINKS (FROM source TO destination
              PICTURES (arrowhead AT destination));

EDGE_TYPE control_flow
    NODES (source      AT (0, 0),
           control     AT (FlowL/2, 20),
           destination AT (FlowL, 0))
    LINKS (FROM source TO destination
              PROPERTIES (dashing = "(2, 4)")
              PICTURES (arrowhead AT destination));

CONSTANT RepFlowL = 130;      /* Replicated flow edge length */

EDGE_TYPE replicated_flow
    NODES (source      AT (0, 0),
           destination AT (RepFlowL, 0))
    LINKS (FROM source TO destination
              PICTURES (arrowhead AT destination));

CONSTANT ResultsL = 90;

PICTURE_TYPE triang_head
    LINE TO (-20, -10)
    LINE TO (-20, 10)
    LINE FROM (-20, -10) TO (-20, 10);

EDGE_TYPE results_in
    NODES (creator AT (0,0),
           object AT (ResultsL, 0))
    LINKS
      (FROM creator TO object
         PICTURES (triang_head AT object)
         PROPERTIES (cutoff_2 = 20));

DIAGRAM_TYPE functional_model
    PROPERTIES (x_size = 440, y_size =340);
```