The following paper was originally published in the
Proceedings of the USENIX 1996
Conference on Object-Oriented Technologies
Toronto, Ontario, Canada, June 1996.

Software Composition with Extended Entity-Relationship Diagrams

Pornsiri Muenchaisri and Toshimi Minoura
Oregon State University

Software Composition with Extended Entity-Relationship Diagrams

Pornsiri Muenchaisri and Toshimi Minoura

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602
muenchp@cs.orst.edu
minoura@cs.orst.edu

Abstract

We introduce a compositional approach to application software construction. In our approach, an *extended entity-relationship diagram* (EERD), which represents the component types and the relationship types within an application domain, is used as a template of executable programs in that application domain. As we use *structural active objects* as the components of a program, we can obtain an executable program simply by connecting them as dictated by an EERD. Furthermore, the graphical editor in the proposed software development environment uses EERDs as menus in constructing application software. An EERD used as a menu can enforce legitimate patterns of relationships among components, besides that they provide an intuitive view of available components and possible relationships among them.

1 Introduction

In the late 1960s practitioners and researchers began to discuss a software crisis in developing a large software system with existing methods [Pree87, Gold95, Lamb87, Lewi91, Somm92]. In order to improve productivity and quality in software development, many new techniques and methods to manage complexity of software were developed, but the problem has remained unresolved [Booc94], and serious research still continues in this area.

The *entity-relationship* (ER) approach was first proposed by Chen [Chen76]. Since then it has been extensively used in designing schemas for database systems [Elma89] and to represent the structures of systems in systems analysis [Chen83]. An ER diagram not only provides an intuitive view of an application, but it is also possible to generate automatically a relational schema from it [Teor82]. The ER approach turned out to be very effective in the areas of data management and systems analysis.

In the area of general software development, the object-oriented (OO) approach was one of the most successful [Cox86, Gold95]. To represent the structures of OO software, *Object Modeling Technique* (OMT) [Rumb91], the *Booch method* [Booc94], and the *Fusion method* [Cole94] were developed. Although the object models are extensions of ER diagrams, they incorporate various additions including behavior modeling. Also skeleton code can be generated from an object model.

Composing application software from software components, as other industrial products are produced from their components, has been an aim of many researchers [Cox86, Budd91, Nier91, Jaco92, Mino93b, Seli94, Shaw95b]. By using well-tested software components, we can reduce the development time and enhance the quality of application software. Cox presented the notion of *Software-ICs*, which are reusable software components produced from components from other applications [Cox86]. Nierstrasz et al. developed a *visual scripting* tool called Vista [Nier91, Nier92]. With this tool, a user can interactively construct an application from pre-packaged, plug-compatible software components by direct manipulation and graphical editing.

The *structural active-object system* (SAOS) approach was an attempt to generate executable programs from their graphical representations [Mino93a, Mino93b, Mino93d]. To make a program composed from interconnected components executable, composable active objects, called *structural active objects*, are used as building blocks of software. Structural active objects encapsulate not only data but also control and are better modularized than ordinary objects. Structural active-objects are interconnected with each other through their *structural interfaces*, and composite active-objects thus constructed are hierarchically composed. This mechanism of component composition is referred to as *structural and hierarchical object composition* (SHOC).

Schappert, Sommerlad, and Pree proposed software components accompanied by an *active cookbook* [Scha95]. An active cookbook provides recipe-like on-line guides to help the user compose working software from the framework components. How components can be composed is determined by the *relations* defined among them.

In this paper, we present the design of a software development environment, called *Entity-Relationship Software Development Environment* (ERSDE), for creating executable software automatically. The environment is based on the ER approach with structural active objects. The environment uses an *extended entity-relationship diagram* (EERD) (*domain-specific schema*) as a menu for a graphical editor as well as as a template of executable programs. A programmer can see in a EERD the available entity types and the patterns of the relationships (possible connections) among them. She can compose an application software by instantiating entities from the entity types in the EERD and then by connecting them following the patterns of relationships specified in the EERD. Since entities are implemented as active-objects, the application is executable as soon as entities are instantiated and interconnected.

In Chapter 2, we provides an overview of our approach by using a simple example. Chapter 3 gives a description of ERSDE. In Chapter 4, we demonstrate effectiveness of our approach by applying it to several application domains. Chapter 5 concludes this paper.

2 Overview

In this section we give an overview of our approach with a simple example. Fig. 1 shows a tank system consisting of tanks, valves, and pumps. A tank contains some kind of liquid, a pump makes liquid flow, and a valve controls the amount of liquid that flows through it. The liquid flows from left to right from tanks to other tanks through valves and pumps.

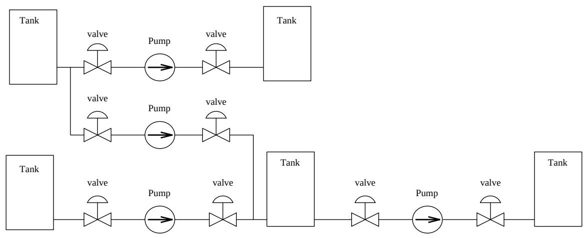

Figure 1: A tank system.

Fig. 2 is a conventional entity-relationship (ER) diagram for tank systems consisting of tanks, valves, and pumps. The input end of a pump can be connected to the output end of a valve, and the output end of a pump to the input end of a valve. The input end of a tank can be connected to the output ends of many valves, and the output end of a tank to the input ends of many valves.

A simple way to enhance understandability of an ER diagram is to replace the rectangular representation of entity types by their iconic

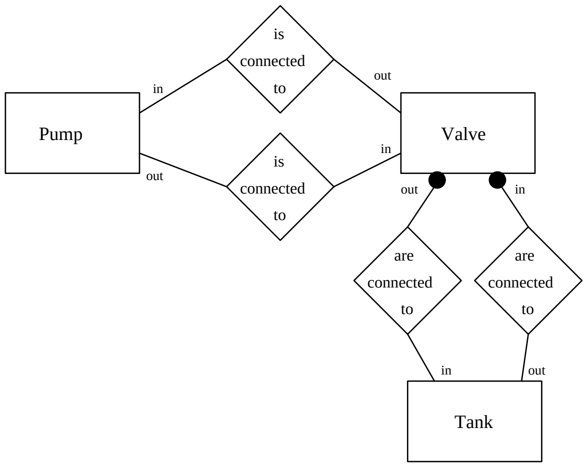

Figure 2: A conventional entity-relationship diagram for tank systems.

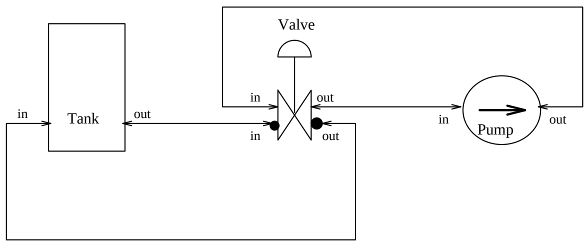

Figure 3: Extended entity-relationship diagram with iconic entity types.

representations. An iconic entity type looks like a real entity. For example, a picture of a valve can be used to represent an entity type `Valve`. Intuitiveness of a domain-specific schema is enhanced with iconic entity types. Fig. 3 shows an *extended entity-relationship diagram* (EERD) of the tank system represented in this way. In this diagram, relationship types are represented by arrows.

However, the representation as shown in Fig. 3 has the following problem. The arrow between the output end of the valve type and the input end of the pump type and the arrow between the output end of the pump type and the input end of the valve type may mislead us to believe that there is a circular connection between a valve and a pump. Similarly, it looks like there is a circular connection between a tank and a valve. This problem will make the diagram confusing or at least unattractive.

To solve the problem described above, we introduce *proxy entity types*. A proxy entity type, which are drawn with dashed lines, is equivalent to the original entity type, and all the connections made to it have the same effect as they were made to the original one. Fig. 4 shows the EERD for tank systems with a proxy of the valve type and a proxy of the tank type. In this way, we can eliminate circular connections which may not exist at the entity level.

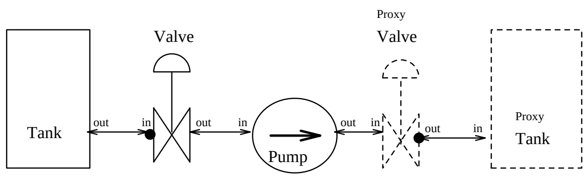

Figure 4: Extended entity-relationship diagram with proxy of entity types.

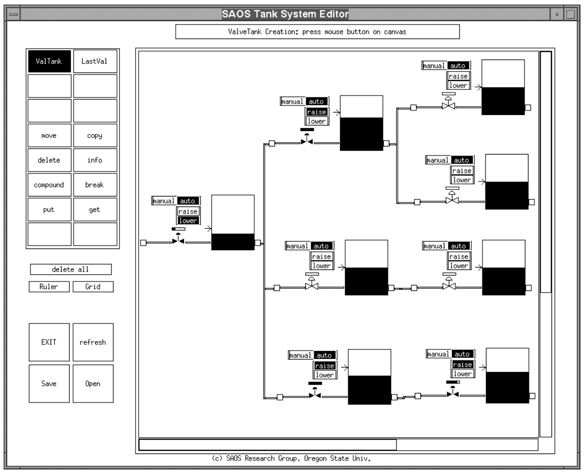

Figure 5: An application of a tank system.

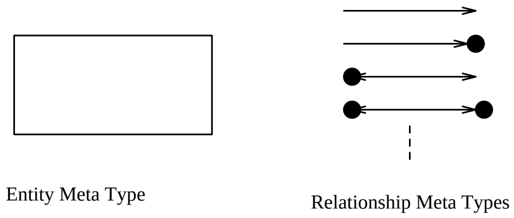

Figure 7: Simplified meta schema.

3 Entity-Relationship Software Development Environment

In this section we present a general framework of a software development environment based on our approach. As shown in Fig. 6, the environment, which we call ERSDE (Entity-Relationship Software Development Environment), consists of three major parts: the *meta schema*, the *schema editor*, and the *application editor*. The major characteristic of ERSDE is that it can be used to develop executable application software in different application domains by component composition.

The meta schema, which contains the *entity metatype* and the *relationship metatypes* as shown in Fig. 7, is used as a template for creating *domain-specific schemas*. When an entity type is created from the entity metatype, the rectangular representation of an entity type may be replaced by an *iconic* representation of the entity type. The relationship metatypes indicate the *cardinality ratios* (*one-to-one*, *one-to-many*, *many-to-one*, *many-to-many*) and the *directions of access*. Additional notations used by the meta schema are described in the next subsection.

A domain-specific (application-specific) schema is an EERD consisting of entity types and relationship types among them. We have already shown an example of an EERD for tank systems in Fig. 4. The schema editor is used to construct domain-specific schemas by instantiating the entity metatype and the relationship metatypes in the meta schema. The schema editor can create and modify EERDs in different application domains.

We can construct applications (instance diagrams) in each application domain with the application editor. This application editor uses the EERD in each application domain as an editor menu. The application editor allows applications to be composed in different application domains by switching the EERD used as its menu.

A SAOS application editor can create executable programs by component composition. However, each SAOS editor is domain-specific, and its menu is a list of items, not an EERD. The SAOS application editor for tank systems is shown in Fig. 5.

In the next three subsections, we describe more details of the three major parts of ERSDE.

3.1 Meta Schema

The meta schema provides notations for creating entity types and relationship types for domain-specific schemas. We adopt some conventional notations and propose some new ones for the meta schema. The meta schema shown in Fig. 8 provides notations for the *entity metatype*, *entity subclassing*, *entity composition*, *relationship metatypes*, and *proxy entity metatype*. The first four notations are extensively used in many object-oriented design methods including OMT, Booch, Fusion [Rumb91, Booc94, Cole94]. However, in our approach, the arrows representing relationship types indicate the directions of data access. A visibility graph of the Fusion method uses an arrow to indicate the direction of data access as we do. However, only one way of data referencing is allowed [Cole94]. The concepts of proxy entity types and relationship representation by proximity, which we describe later, are new.

The entity metatype is the type for the entity types in EERDs. An entity type is instantiated from this entity metatype. Although the generic notation for an entity type is a rectangle, it can be replaced by an iconic representation in a domain-specific schema.

Entity subclassing is a mechanism for an en-

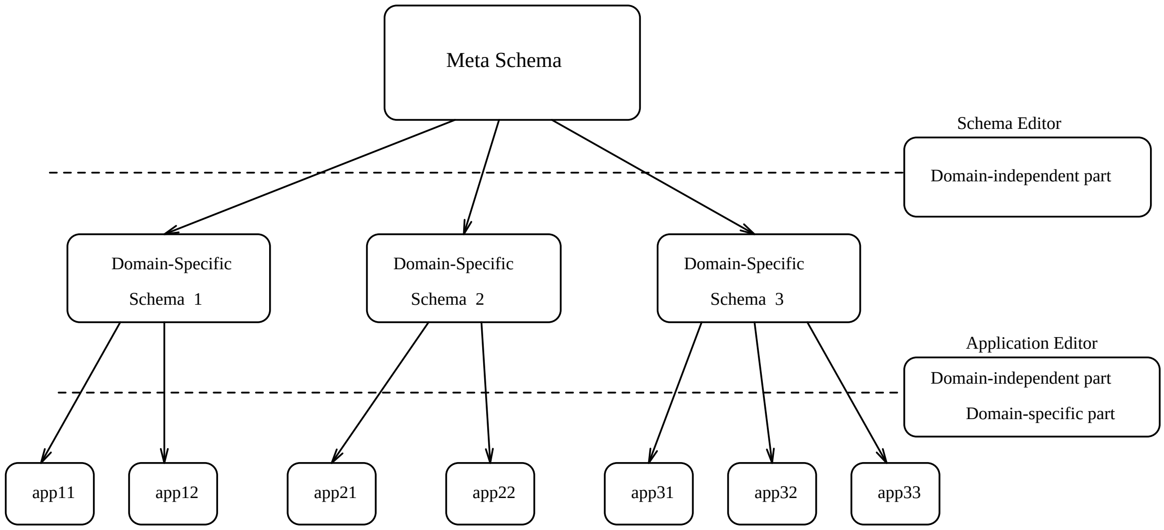

Figure 6: Structure of the ERSDE software development environment.

tity type to inherit some characteristics from another entity type. We use the same notation for entity subclassing as is used by OMT, where entity subclassing is represented by a small triangle in the middle of the line that connects a parent entity type to its child entity types.

Entity composition is a mechanism to allow hierarchical composition of an entity from its component entities. A *composite entity* is an entity created by entity composition. A *composite entity type* is represented by a rectangle with double dashed outlines. Such a rectangle encloses the entity types of its members.

The relationship metatypes in the meta schema are templates for creating relationship types in EERDs, where entity types are connected with other entity types by relationship types. A relationship metatype is a *unidirectional* or *bidirectional* arrow with small filled-circles at its ends. A small filled-circle ($\bullet$) means "many". The tag attached to a relationship type represents the attributes of the relationship type. The semantic direction of a relationship is normally from left to right or from top to bottom. In this paper, we refer to the source of a relationship as its left-side entity, and the destination as its right-side entity. There are four possible combinations of these small filled-circles.

1. *One-to-one*: The relationship metatype is one-to-one if there is no small filled-circle at either end of the arrow.

2. *One-to-many*: The relationship metatype is one-to-many if there is a small filled-circle at the right end of the arrow.

3. *Many-to-one*: The relationship metatype is many-to-one if there is a small filled-circle at the left end of the arrow.

4. *Many-to-many*: The relationship metatype is many-to-many if there are small filled-circles at both ends.

The direction of an arrow indicates that of data access. The access direction of data may be different from the semantic direction. There are three possible cases for the direction of an arrow.

1. *Right-end access* (E1 $\longrightarrow$ E2): If an arrow head is at the right end of the arrow, an instance of E1 can access an instance of E2, but the instance of E2 cannot access the instance of E1.

2. *Left-end access* (E1 $\longleftarrow$ E2): If an arrow head is at the left end of the arrow, an instance of E2 can access an instance of E1,

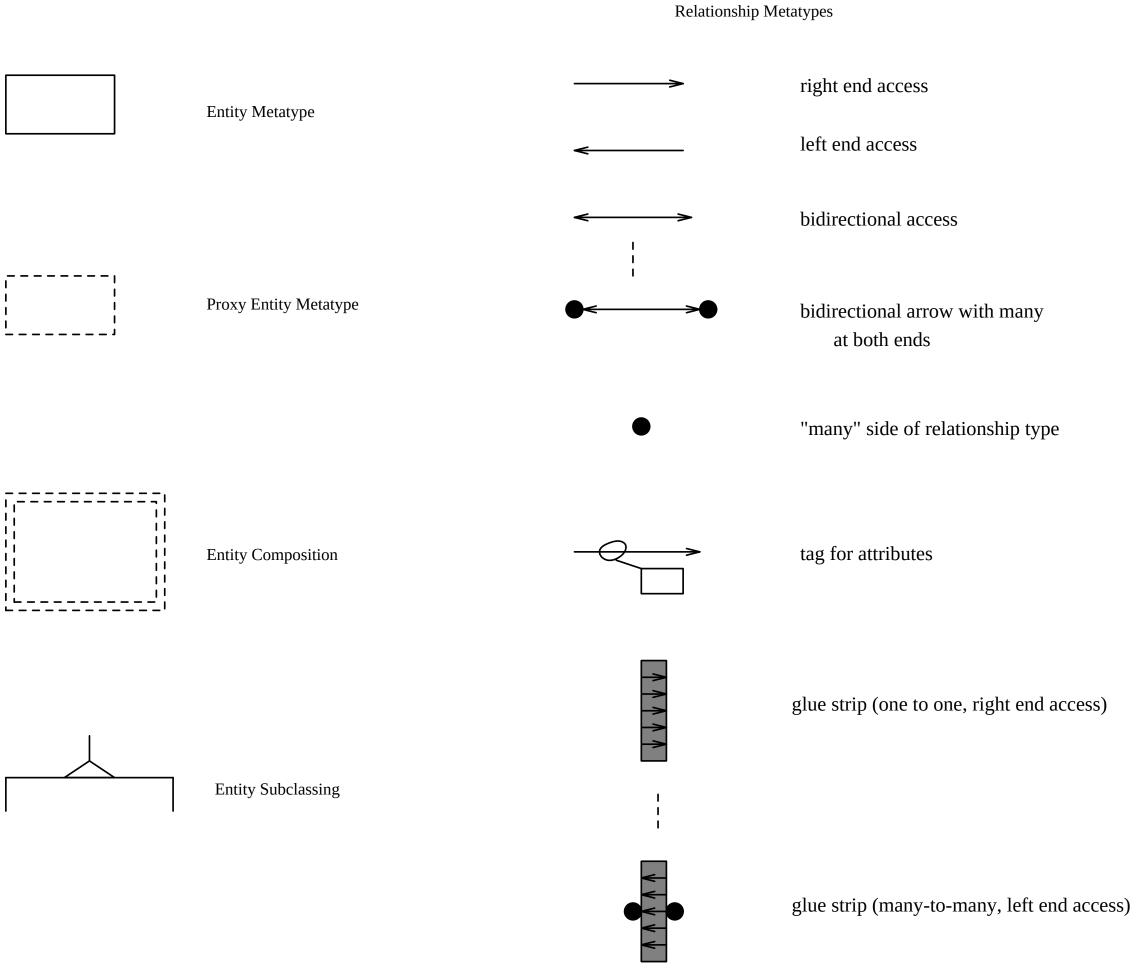

Figure 8: Meta schema.

but the instance of E1 cannot access the instance of E2.

3. *Bidirectional access* (E1 $\longleftrightarrow$ E2): If arrow heads are at both the right and left ends, an instance of E1 can access an instance of E2, and the instance of E2 can access the instance of E1.

All possible combination of cardinality ratios and access directions for the relationship metatypes are given in Fig. 9. Fig. 9(a) shows the one-to-one relationship metatype where an instance of E1 can access an instance of E2, but the instance of E2 cannot access the instance of E1. Fig. 9(c) shows the one-to-many relationship metatype where an instance of E1 can access multiple instances of E2, but these instances of E2 cannot access the instance of E1. Fig. 9(g) shows the many-to-many relationship metatype where an instance of E1 can access multiple instances of E2, but these instances of E2 cannot access the instance of E1. Fig. 9(l) shows the many-to-many relationship metatype where an instance of E1 or E2 can access the related instances of E2 or E1, respectively.

In current object-oriented programming languages, relationships are implemented by pointers. Since pointers cannot carry any attribute information, we must store attribute information of relationships in entities. For a one-to-one relationship type, we can move the attributes of the relationship type to the entity type at either the left or right side of the relationship type. Fig. 9(a), Fig. 9(b), and Fig. 9(i) are examples of one-to-one relationship types whose attributes can be moved to entity type E1 or E2. For a one-to-many or many-to-one relationship type, we can move the attributes of the relationship type to the entity type at the "many" side of the relationship type. Fig. 9(c), Fig. 9(f), Fig. 9(j) are examples of one-to-many relationship types whose attributes can be moved to entity type E2. Fig. 9(d) Fig. 9(e), Fig. 9(k) are examples of many-to-one relationship types whose attributes can be moved to entity type E1. For a many-to-many relationship type, we cannot move the attributes of the relationship type to entity type E1 or E2. Therefore, a many-to-many relationship with attributes must be implemented as an entity. Fig. 9(g), Fig. 9(h) and Fig. 9(l) are examples of many-to-many relationship types.

We now explain the reasons why proxy entity-types are introduced. The proxy entity types are designed to make an EERD easy to understand. Proxy entity types are equivalent to their original entity types. The following problems are examples that proxy entity types can solve.

1. (Multiple Sheet Problem) When a system is large, multiple sheets are needed to show all the required entity types and relationship types. Then there should be a way to refer to entity types in other sheets. From one sheet we can refer to an entity type given on another sheet with a proxy entity type.

2. (Circular Connection Problem) This problem occurs when some entities are connected to other entities of the same type. In this case, a chain of relationship types originates from and ends at the same entity type. We have already illustrated this problem and its solution in Fig. 3 and in Fig. 4, respectively.

3. (Multiple Component Problem) This problem occurs when a composite entity type includes multiple occurrences of one entity type as its components. Fig. 10 shows a standard ER diagram for an entity type `Car` which is a composite type consisting of four occurrences of the entity type `Wheel` and one occurrence of the entity type `Body`. The fact that a car has four wheels is not intuitively represented. The EERD given in Fig. 11, on the other hand, shows the composite entity type `Car` by using proxy entity types.

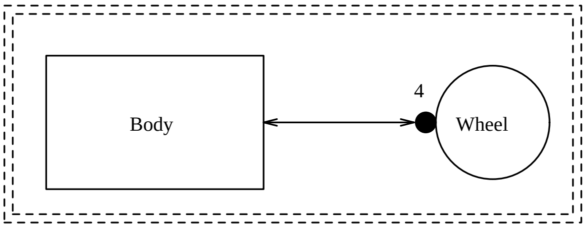

Figure 10: A car as a composite entity type.

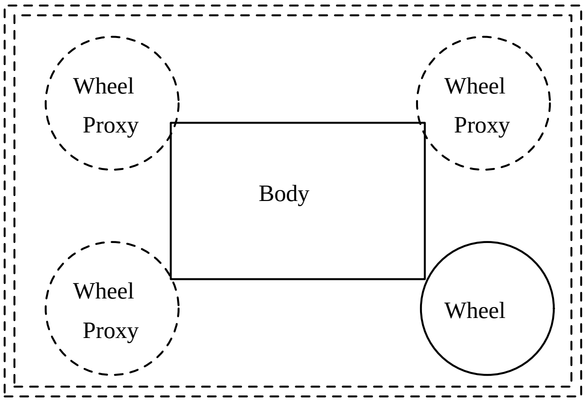

Figure 11: A car as a composite entity type with proxies.

Although the generic notation for a proxy entity metatype is a dashed rectangle, it can be replaced by an iconic representation in a domain-specific schema.

3.2 The Schema Editor

The ERSDE provides a graphical schema editor for creating, deleting, and moving entity types in EERDs. We use the schema editor to build domain-specific schemas. The domain-specific schemas display entity types and relationship types among them.

The schema editor is a (general) domain-independent graphical editor for creating and manipulating graphical representations of entity

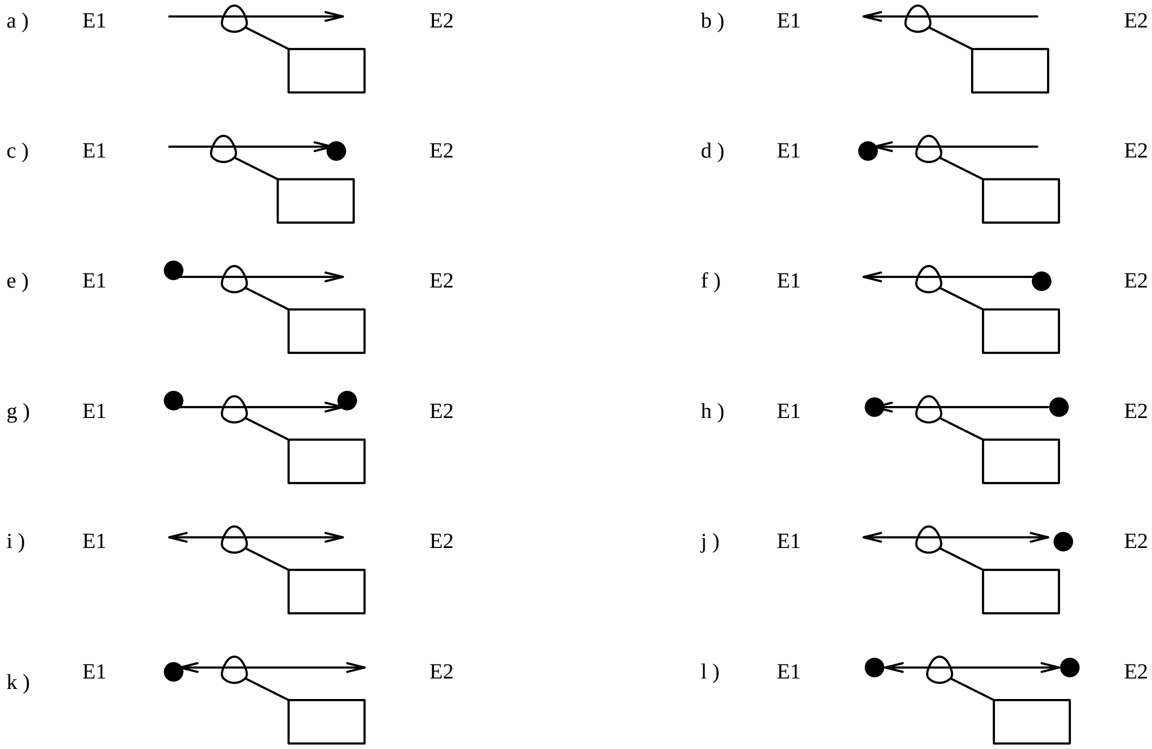

Figure 9: Relationship metatypes between two entity types.

types and relationship types in domain-specific schemas.

There are four major steps in constructing a domain-specific schema. First, we use the schema editor to create entity types. Second, we connect entity types to other entity types by relationship types. Third, once an EERD is completed, the schema editor generates skeleton code for the entity types and the pointer structures to access related entities according to the direction of data access specified in the domain-specific schema. Fourth, a programmer is responsible for providing behaviors for each entity type.

We use some new ideas in domain-specific schemas: iconic representations of entity types, proxy entity types, and *relationship representation by proximity.*

We can relate entity types to other entity types by placing them closely. This mechanism for creating relationship types is called relationship representation by proximity. Although relationship types shown by this mechanism are semantically not different from those represented by arrows, composite (assembly) entity types shown in this way look more like real entities. We use a *grey glue strip* to represent a relationship type by proximity. The cardinal ratio of a relationship type can be indicated with a small filled-circle within an entity type on the "many" side. An EERD for cars using this notation among its component types is shown in Fig. 12. A `Car` has one `Driver` and multiple `Passenger`s.

3.3 Application Editor

We can use the application editor to construct applications from a domain-specific schema (EERD). In composing an application, the domain-specific schema is used as the menu of the application editor to instantiate entity types and connect them in compliance with the connectivity styles in the EERD. When used as a menu of a graphical editor, an EERD is more effective than a conventional editor menu since it

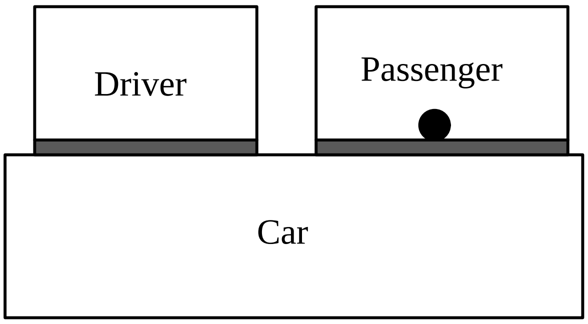

Figure 12: The EERD for a car, a driver and passengers.

can show not only entity types but also possible relatioships between entities.

The application editor consists of the domain-independent part and the domain-dependent part. The domain-independent part is a general graphical editor used to create, and move instances of entity types and connections among them. The domain-dependent part handles application-specific editing operations. For the schema editor, an EERD can be regarded as an instance diagram and the meta schema as a schema.

If each entity is an active object, the application can be executed once the entities are connected. The application allows us to construct applications in different layouts, and to move, delete, and edit components interactively.

4 More Examples

In this section we demonstrate the generality of our approach by applying it to examples in five different domains. For each domain, we provide the conventional ER diagram, the EERD, and an example of application. Our objective is to show that EERDs are more intuitive than conventional ER diagrams.

4.1 Queuing Systems

We first consider queuing systems. Fig. 13 shows a queuing system consisting of a *generator*, three *queues*, and two *processors*. The generator produces jobs, and a queue holds jobs. A processor removes a job from its input queue, processes it, and passes the job to its output queue.

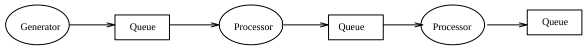

Figure 13: A queuing system.

Fig. 14 is a conventional entity-relationship (ER) diagram for queuing systems. The output end of a generator can be connected to the input end of a queue. The input end of a processor can be connected to the output end of a queue, and the output end of a processor to the input end of a queue. Multiple processors can be connected to a single queue for job input and output.

The EERD for the same application domain is shown in Fig. 15. A proxy of the type `Queue` is used to avoid a circular connection between the type `Processor` and the type `Queue`. The EERD explicitly shows the direction of the data access. An example of a queuing system application instantiated from the EERD is shown in Fig. 16.

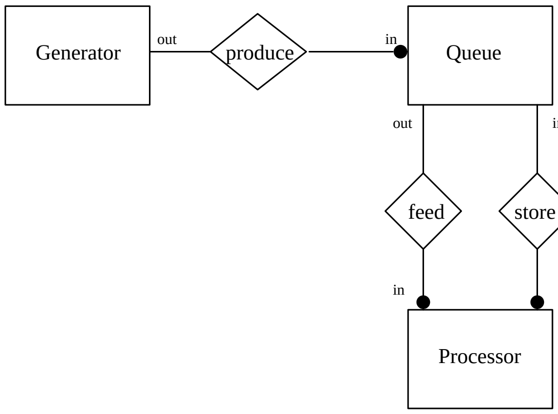

Figure 14: A conventional entity-relationship diagram for queuing systems.

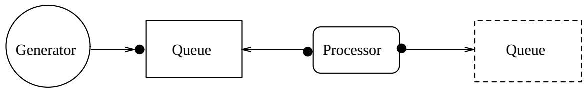

Figure 15: The extended entity-relationship diagram for queuing systems.

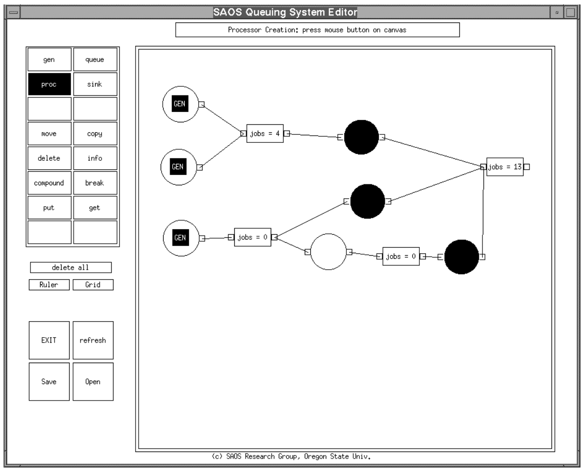

Figure 16: A queuing system application.

4.2 Wide Area Network Systems

We now consider simulation of wide area network (WAN) systems. A simulator for a WAN system consists of *host*s, *link*s, and a *recorder*. A host communicates to another host via a link. The load incurred on a link is referred to as *usage*, which we treat as a component of the link. A recorder displays the usage of a link.

Fig. 17 shows a conventional ER diagram for WAN systems. The input end of a link is connected to a host, and the output end of the link is connected to another host. Multiple links can be connected to a host. A recorder is connected to the usage component of a link.

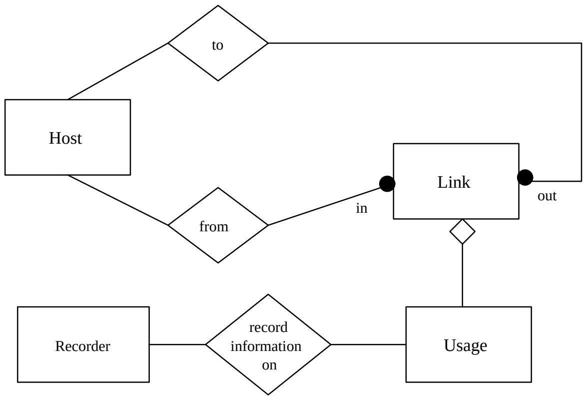

Figure 17: A conventional ER diagram for wide area network systems.

The EERD for WAN systems is shown in Fig. 18. A link is a relationship which has attribute `usage`. The attribute `usage` is an entity. A programmer can see how a host can be connected to another host via a link more clearly in the EERD than in the conventional ER diagram. Note that a relationship with attributes must be implemented as an object in a current OO language. An iconic representation is used for type `Recorder`. Fig. 19 shows an example of a WAN simulator.

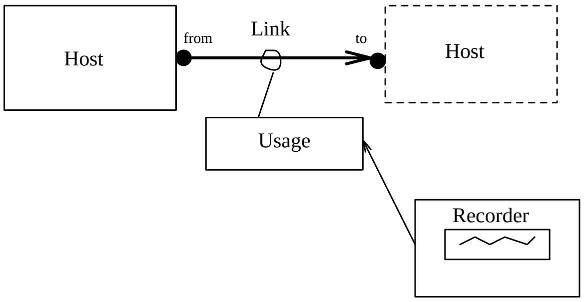

Figure 18: The EERD for wide area network systems.

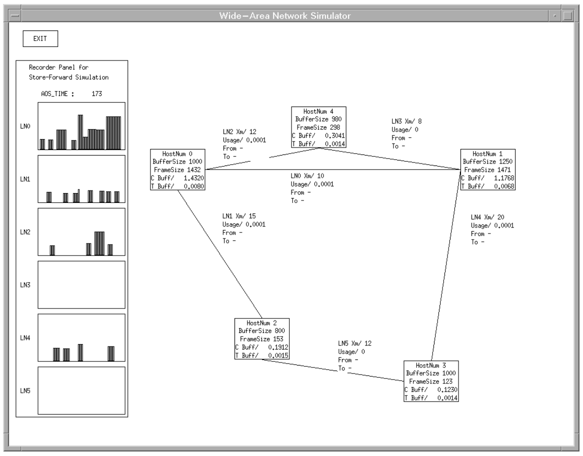

Figure 19: A wide area network system application.

4.3 Local Area Network Systems

Let us now consider simulators of local area networks (LANs). A LAN simulator consists of ethernet *cables*, *stations*, and *repeaters*. A cable is divided into multiple *cable segments*. A station is connected to a cable segment, and a repeater is placed between two cable segments.

The conventional ER diagram for LANs is shown in Fig. 20. A cable consists of multiple cable segments. The left end of a cable segment may be connected to the right end of another cable segment, and its right end to the left end of still another cable segment. One end of a repeater is connected to a cable segment and another end to another cable segment. A station is connected to a cable segment.

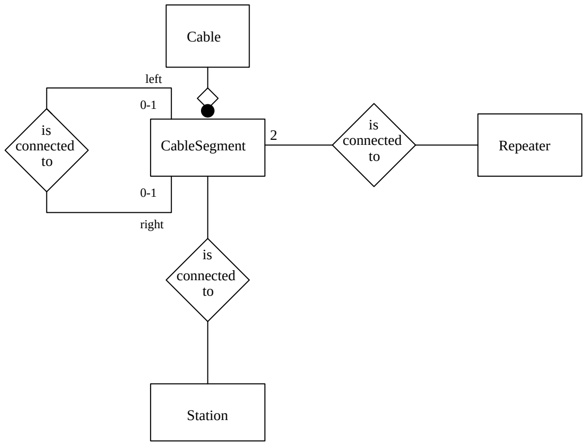

Figure 20: A conventional ER diagram for local area network systems.

The EERD for LANs is shown in Fig. 21. The fact that a cable consists of an array of cable segments is graphically shown with dashes (- - -). Furthermore, a connection of a cable segment to another one at its end is shown by a grey (*glue*) strip representing a relationship. A grey strip is one way to represent a relationship by graphical proximity. Fig. 22 shows an example of a LAN simulator.

4.4 A Digital Circuit Simulator

We now apply our approach to digital circuits. A digital circuit consists of various (`AND`, `OR`, `NOT`, `NAND`, `NOR`) *gates*, *signal generators*, *clock generators*, and *recorders*.

Fig. 23 shows the conventional ER diagram for digital circuits. Each of a `SignalGen` (signal generator), a `clock`, and a `Gate` has an attribute `output` of type `Signal`. A recorder takes a `Signal` as its input. A `Gate` is a `1-Gate`, a

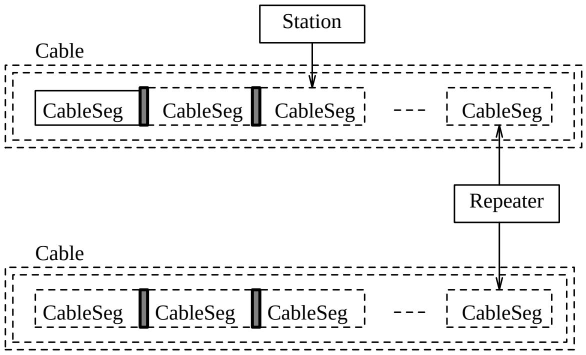

Figure 21: The EERD for local area network systems.

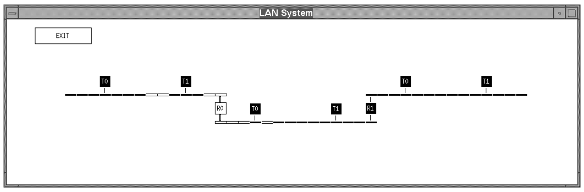

Figure 22: A local area network system application.

`2-Gate`, or an `M-Gate`. A `1-Gate` takes one input of type `Signal`, and `2-Gate` takes two inputs of type `Signal`. Type `NOT` is a subtype of type `1-Gate`. We consider `AND`, `OR`, `NAND`, and `NOR` gates as two-input gates, and we consider `M-AND` and `M-OR` gates as multiple-input gates.

The EERD for the digital circuits is shown in Fig. 24. Type `Signal Generator`, `Clock`, and `Gate` are shown as composite entity types. Proxies of type `Signal` are used as input signals of type `Recorder`, Type `1-Gate`, and type `2-Gate`. Type `NOT`, type `AND`, type `OR`, type `NAND`, type `NOR`, type `M-AND`, and type `M-OR` use their standard iconic representations. Fig. 25 shows a D-latch circuit as an example of a digital circuit.

4.5 Automatic Guided Vehicle Systems

We finally consider simulation of automatic guided vehicle systems (AGVs) used in automated factories. An AGV simulator consists of *track segments*, *intersections*, *stations*, *AGVs*, *jobs*, and *routes*. An AGV travels on the track

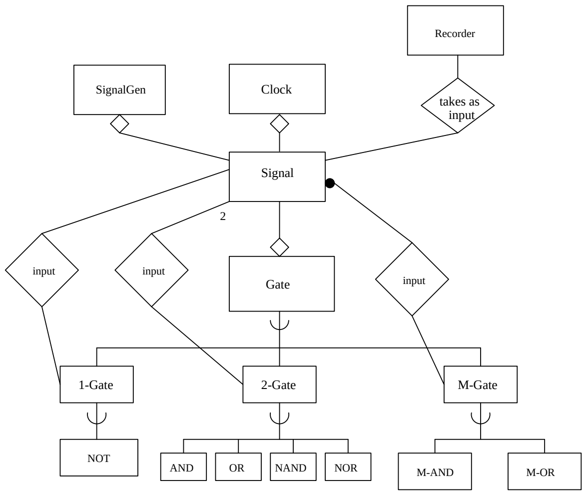

Figure 23: A conventional ER diagram for digital circuits.

segments carrying multiple jobs and following a route. Each job is picked up by an AGV at a station and delivered to another station. An intersection is a connection point of one or more track segments. A track segment must be connected to an intersection at each end. An intersection can also be connected to a station. At any one time, only one AGV can occupy an intersection.

Fig. 26 shows the conventional ER diagram for AGV systems. The diagram shows all possible connections among entity types.

The EERD for AGV systems is shown in Fig. 27. Proxies of `AGV`, `Station`, `Job`, `TrackSeg` (track segment), and `Intersection` are used to make the diagram intuitive. Relationship representations by proximity (grey glue strips) are used between `AGV` and `Station`, `Station` and `Intersection`, `Job` and `AGV`, `AGV` and `TrackSeg`, `AGV` and `Intersection`, `Job` and `Station`, and `Station` and `Intersection`. `AGV` on `Station`, which in turn is on `Intersection`, means that an `AGV` can be at a `Station`, which in turn is adjacent to an `Intersection`. An `AGV` can also be in an `Intersection`. A small-filled circle inside type `Job` on type `AGV` indicates that the relation ships between `Job` and `AGV` are many-to-one, implying that a `AGV` can carry multiple `Job`s. The left end of a `TrackSeg`

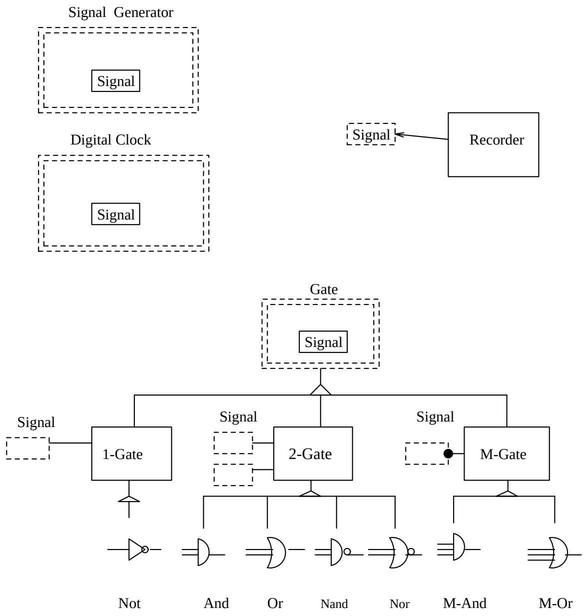

Figure 24: The EERD for digital circuits.

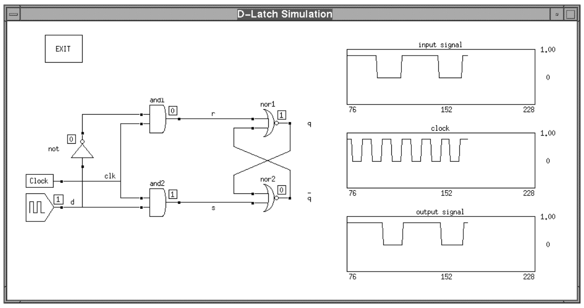

Figure 25: A D-latch circuit.

can be connected to an `Intersection` and the right end of a `TrackSeg` can be connected to another `Intersection`. The fact that multiple `TrackSeg`s can be connected to an `Intersection` is shown by small filled-circles within `TrackSeg`. Fig. 28 shows an example of a AGV simulator.

5 Conclusions

We presented a software development environment for composing application software from components. The environment uses *structural active objects* as components, and it includes

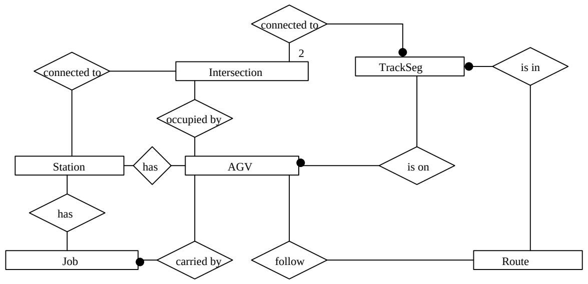

Figure 26: A conventional ER diagram for AGV systems.

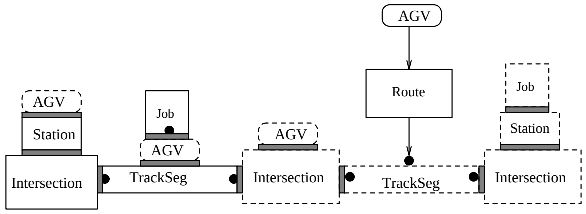

Figure 27: The EERD for auto guided vehicle systems.

three main parts: the *meta schema*, the *schema editor*, and the *application editor*.

The meta schema provides notations for extended entity-relationship diagrams (EERDs). We can use the schema editor to create domain-specific schemas as EERDs by creating entity types and then connecting them with relationship types. Once the EERD for an application domain is complete, the schema editor can generate skeleton code for the entity types and the pointer structures implementing the relationship types.

An application can be composed by instantiating entities from entity types in an EERD and then connecting those entities according to the permissible relationship patterns shown in the domain-specific schema. If behavior definitions have been attached to entity types manually with the schema editor, application software created from the EERD becomes executable.

The ER approach has been used extensively in the data management area and in the system analysis area. We demonstrated that we can construct an executable software program from their components by connecting them exactly

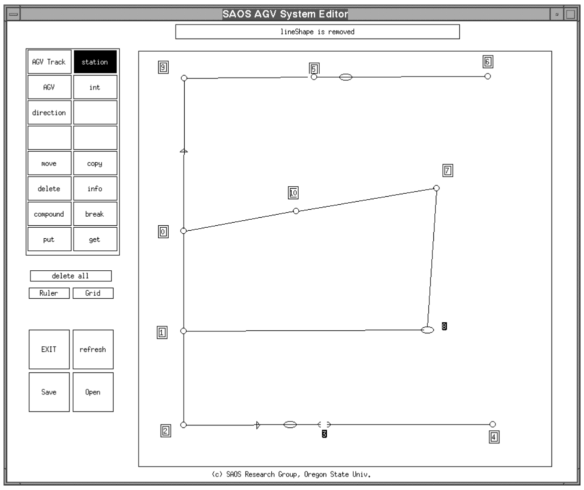

Figure 28: An application of auto guided vehicle systems.

as indicated by an extended ER diagram when structural active objects are used as software components.

To use EERDs as templates to compose applications, we introduced some new ideas and notations, e.g., iconic entity types, proxy entity types, and relationship representation by proximity. We then presented the architecture of the entity-relationship software development environment (ERSDE). One new idea is to use an EERD as a menu for a graphical editor. Compared to an ordinary menu, which is simply a list of items that can be created, an EERD can show possible patterns of connections among the entities created. Furthermore, by using iconic representations in the ER diagram, its intuitiveness can be enhanced.

We have implemented several *application-specific* editors that allows us to create application software by component composition. Currently we are developing the schema editor and the application editor so that the application software in different domains can be composed.

References

[Booc94] Grady Booch. *"Object-Oriented Analysis and Design with applications"*, chapter 1, pages 3–26. Benjamin-Cummings, 2 edition, 1994.

[Budd91] Timothy Budd. *"An Introduction to Object-Oriented Programming"*, chapter 1, pages 14–15. Addison-Wesley, 1991.

[Chen76] Peter Pin-Shan Chen. "The Entity-Relationship Model - Toward a Unified View of Data," *ACM TOD* 1(1):9–36 March 1976.

[Chen83] Peter Pin-Shan Chen. "A Preliminary Framework for Entity-Relationship Models," *Entity-Relationship Approach to Information Modeling and Analysis* pages 19–28 1983.

[Cole94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, , and Paul Jeremaes. *"Object-Oriented Development: The FUSION method"*, chapter 1-4, 8, 9, pages 1–230. Prentice-Hall, 1 edition, 1994.

[Cox86] Brad J. Cox. *"Object Oriented Programming - An Evolutionary Approach"*. Addison-Wesley, 1986.

[Elma89] Ramey Elmasri and Shamkant B. Navathe. *"Fundamentals of Database Systems"*. The Benjamin/Cummings Publishing Company Inc., 1989.

[Gold95] Adele Goldberg and Kenneth S. Rubin. *"Succeeding with Objects: Decision Frameworks for Project Management"*, chapter 14, page 315. Addison-Wiley Publishing Company, Inc., 1995.

[Jaco92] Ivar Jacobson, Magnus Christerson, Patrik Johnson, and Gunnar Overgaard. *"Object-Oriented Software Engineering: A Use Case Driven Approach"*, chapter 11, page 291. Addison-Wesley, 1992.

[Lamb87] David Alex Lamb. *"Software Engineering: Planning for Change"*, chapter 1, pages 1–6. Prentice Hall, 1987.

[Lewi91] T. G. Lewis. *"CASE: Computer-Aided Software Engineering"*, chapter 1, pages 11–25. Van Nostrand Reinhold, 1991.

[Mino93a] T. Minoura, S. Choi, and R. Robinson. "Structural active-object systems for manufacturing control," *Integrated Computer-Aided Engineering* 1(2):121–136 1993.

[Mino93b] Toshimi Minoura, Shirish S. Pargaonkar, and Kurt Rehfuss. "Structural Active Object Systems for Simulation,". In *OOPSLA'93 proceedings*, pages 338–355. ACM, October 1993.

[Mino93d] T. Minoura and Sungoon Choi. "Structural active-object systems fundamentals,". Technical Report 93-40-04, Dept. of CS, Oregon State University, 1993.

[Nier91] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, and Marc Stadelmann. "Objects + Scripts = Applications," *In Proceedings, Esprit 1991 Conference* pages 534–552 1991.

[Nier92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. "Components-Oriented Software Development," *Communication of ACM* 35(9):160–165 Sept 1992.

[Pree87] Roger S. Pressman. *"Software Engineering: A Practitionaer's Appproach"*, chapter 1, pages 1–20. McGraw-Hill, 2 edition, 1987.

[Rumb91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *"Object-Oriented Modeling and Design"*, chapter 1-12, pages 1–277. Prentice-Hall, 1 edition, 1991.

[Scha95] Albert Schappert, Peter Sommerlad, and Wolfgang Pree. "Automated Support for Software Development with Frameworks,". In *Symposium on Software Reusability*, pages 123–127. ACM SIGSOFT, April 1995.

[Seli94] Bran Selic, Garth Gullekson, and Paul T. Ward. *"Real-Time Object-Oriented Modeling"*. John Wiley and Sons, Inc., 1994.

[Shaw95b] Mary Shaw, Robert DeLine, Daniel V. Klien, Theodore L. Ross, David M. Young, and Gregory Zelesnik. "Abstractions of Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering* 21(4):314–335 April 1995.

[Somm92] Ian Sommerville. *"Software Engineering"*, chapter 1, 11, 16, 17, 18. Addison-Wiley, 4 edition, 1992.

[Teor82] Toby J. Teorey and James P. Fry. *"Design of Database Structures"*. Prentice Hall, 1982.