AN ABSTRACT OF THE THESIS OF

Pornsiri Muenchaisri for the degree of Doctor of Philosophy in Computer Science presented on November 14, 1997.

Title: Software Composition with Extended Entity-Relationship Diagrams.

Redacted for Privacy

Abstract approved: _____

Toshimi Minoura

I introduce a *compositional approach* to application software development. In this 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 if those components are instantiated and interconnected as dictated by an EERD. Furthermore, the graphical editor in the proposed software development environment, *entity-relationship software development environment* (ERSDE), uses EERDs as menus in constructing application software. An EERD used as a menu can enforce legitimate patterns of relationships among components, in addition to providing an intuitive view of available components and possible relationships among them.

Two experiments were conducted in order to compare the effectiveness between EERDs and class diagrams of Object Modeling Technique (OMT) and between the ERSDE and the menu-based Structural-Active Object System (SAOS) graphical editors. From these experiments, we obtained the following results.

1. A significant proportion of the subjects who used EERDs to compose certain applications did so correctly, while only a small propotion of the students who used the OMT class diagrams composed these applications correctly.

2. Most of the subjects preferred EERDs to OMT class diagrams as design documents.

3. Although the proportion of the students who composed applications correctly with the ERSDE application editor was larger than the proportion of the students who did so with the menu-based SAOS graphical editors, this difference was statistically not significant.

4. The subjects took significantly longer time to compose applications with the menu-based SAOS editors than with the ERSDE editor.

5. All the subjects preferred the ERSDE application editor to the menu-based SAOS graphical editors as a software development environment.

Software Composition with Extended Entity-Relationship Diagrams

by

Pornsiri Muenchaisri

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed November 14, 1997
Commencement June 1998

Doctor of Philosophy thesis of Pornsiri Muenchaisri presented on November 14, 1997

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Pornsiri Muenchaisri, Author

ACKNOWLEDGMENT

There are many people to whom I am greatly indebted for their assistance and support in completing my Ph.D. program. Although I cannot name them all, a few deserve special mention and recognition.

Firstly, I would like to express my sincere thanks and gratitude to my advisor, Dr. Toshimi Minoura. I deeply appreciate his patience, constant guidance and encouragement throughout my course of study here at the Oregon State University.

Secondly, I would also like to thank my committee members Dr. Christopher Bell of the Civil Engineering Department, Drs. Bella Bose, Timothy Budd, and Curtis Cook of the Computer Science Department for their suggestions and help.

Thirdly, I would like to thank all the faculty members of the Department of Computer Science, Oregon State University, for providing the opportunity to pursue a higher degree here, for affording me an invaluable education, and for allowing me to work as their teaching assistant.

Fourthly, I wish to thank the Ministry of Science and Technology, Thailand, for their financial support.

Fifthly, I would like to thank my host family, Kris Walters, John Atkinson and their children, and my American parents, Dr. Charles and Laura Dailey, for making my stay in Corvallis a pleasant and memorable one.

Sixthly, I would like to thank Dr. Suvimol Sujjavanich, Dr. Sherry Yang, Dr. Bader Almohammad, Dr. Chandra Reddy, and Balakrishnan Maharajapuram for their help and support.

Lastly, I would not have had the strength to overcome the various obstacles I encountered while I studied here, without the constant encouragement and support from my parents, and my sister and my friends both here in Corvallis and in Thailand.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF APPENDICES

LIST OF APPENDICES (Continued)

LIST OF APPENDIX FIGURES

Figure Page

LIST OF APPENDIX FIGURES (Continued)

LIST OF APPENDIX TABLES

SOFTWARE COMPOSITION WITH EXTENDED ENTITY-RELATIONSHIP DIAGRAMS

Chapter 1

INTRODUCTION

In this chapter I identify the problem addressed in my research and present my approach to solve it.

1.1 Problem Statement

In the late 1960s practitioners and researchers began to discuss a software crisis in developing a large software system with existing methods [29, 68, 49, 52, 80, 37, 41]. Since then many new techniques, design methods, and tools have been developed in order to help a programmer create an application easily, but the problem has remained unresolved.

Composing application software from components, as other industrial products are produced, has been an aim of many researchers [20, 14, 64, 42, 58, 76, 78, 50]. By using well-tested software components, we can reduce the development time and enhance the quality of application software. However, none of the current software-component composition methods use *patterns of relationships* among components in constructing applications effectively. They emphasize only how components should communicate with each other. Furthermore, although some code for class definitions can be generated from a diagram, the generated code is not executable.

For example, *Universal Connector (Unicon)* [77], allows relations among components to be specified by *connectors*. The Unicon system is not an object-based system and it supports only pre-defined components. Vista is a prototype environment for visual software composition. A Vista application is specified in terms of component behaviors, component presentations, and a composition model [63]. A Vista composition model is expressed only in a textual notation. The current Structural Active-Object System (SAOS) graphical editors, which allow applications to be developed by component composition, do not show relationships among components. Furthermore, a separate SAOS graphical editor is needed for each application domain. A SAOS graphical editor for simple queuing systems and a SAOS graphical editor for tank systems, for example, are different.

1.2 Research Approach

In this research, I integrate an object modeling technique, a (visual) software component composition, entity-relationship diagrams, and active objects as a software development environment. This approach enables application software in different domains to be composed by pick and place with one software environment.

I present a software component composition methodology and tools that support the methodology. The tools support some stages of software life cycle such as design, implementation, operation, and testing.

Although a class diagram of the Object Modeling Technique (OMT) allows us to specify possible patterns of relationships among instances of classes, information on how components are related with each other is not explicitly used when an executable application is created. I believe relationship patterns should be included in the template from which applications are constructed. Such a template will provide an intuitive view of the patterns of relationships in an application domain. When this

template is graphically represented and is used as a menu of a graphical editor, its user can easily comprehend the application domain and construct an application.

I restrict the research to object composition instead of functional (or relational), procedural, or process composition, so that we can create realistic applications by visual composition. Furthermore, I want to be able to create executable programs interactively when active components are provideed as editor components.

I designed and implemented a prototype software development environment called the *Entity-Relationship Software Development Environment* (ERSDE), which follows the software component composition methodology for creating executable software automatically. 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 an EERD the available entity types and the patterns of the relationships (possible connections) among them. She can compose an application 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.

I have conducted two experiments in order to evaluate the approach to application software development. In the first experiment, the effectiveness of EERDs and OMT class diagrams as design documents was compared. I also asked the subjects their preference between the EERDs and OMT class diagrams. In the second experiment, I compared the ERSDE application editor with menu-based SAOS graphical editors in terms of the correctness of applications composed and the times required to compose them. I then asked the subjects their preference between the ERSDE application editor and SAOS graphical editors as a software development environment.

I restrict this research only to visual composition of concurrent systems. The environment is not meant to be a general purpose software development environment. The basic building blocks of each application are objects. In other words, the focus of this research is to find a software development methodology that allows concurrent applications to be constructed by component composition.

1.3 Thesis Plan

The rest of this thesis is organized as follows. In Chapter 2, I discuss the background of this research and discuss related works. Chapter 3 presents the architecture of the ERSDE. In Chapter 4 I explain how to use the ERSDE and also demonstrate the feasibility of the ERSDE by showing three applications created with it. The implementation of the ERSDE is described in Chapter 5. Chapter 6 presents an evaluation of the ERSDE. Chapter 7 presents the conclusions of this thesis and addresses some future research topics.

Chapter 2

RELATED RESEARCH

Application development methodologies have evolved steadily since a software crisis was recognized in the late 1960s. Nowadays software developers no longer build entire programs from basic statements of programming languages. They instead produce application software by using reusable components from libraries and collections of classes.

In this chapter, I discuss research related to this work, including software engineering, the entity-relationship (ER) approach, object-oriented modeling, software component composition, and software development environments.

2.1 Software Engineering

Software engineering is an engineering discipline concerned with the development and maintenance of large software systems. This engineering discipline was formed as a response to the software crisis in the late 1960s. At that time some large-scale software projects incurred large cost overrun and delayed schedules, and others had to be canceled [68, 37, 49, 52, 80, 75, 12, 74].

As the first step of software engineering, a software process (life cycle), which consists of such phases as requirements analysis, system design, unit implementation and testing, system integration and testing, and maintenance, was identified [68, 52, 80]. The *waterfall model* is a software development process where these activities occur in sequential order. The waterfall model is often inadequate for modern

software development because of its extreme emphasis on full documentation of the requirement at an early stage of software development [10].

In incremental development the idea of divide-and-conquer is applied to software development. A system is divided into smaller subsystems, and each of these subsystems is developed separately. When a subsystem is completed, it can be integrated with other completed subsystems. The incremental approach avoids a "Big Bang" in software development [37].

Iterative development is a software development strategy that uses a succession of refinements of a system in order to remove mistakes or to make improvements or add features. A new version of a system is produced at the end of each iteration. A risk analysis is performed for the next iteration in order to discover the risks involved and to devise a strategy to resolve those risks. The quality of the system improves with each iteration.

Goldberg, *et al.* state that incremental and iterative development can be used separately or together [37]. The *spiral model* combines both incremental and iterative development. A user can interact with each version and give feedbacks to developers. Developers do not aim at building the entire system at one time, but they gradually integrate various parts of the system and incorporate requirement changes to the latest version of the system. The system then evolves incrementally and iteratively until a complete product is obtained.

Development methodologies and computer-aided software engineering (CASE) tools are complementary. CASE tools can partially automate a software development methodology and help developers understand the methodology better [54, 80].

2.2 Entity-Relationship (ER) Approach

The *entity-relationship* (ER) approach was first proposed by Chen [15]. Since then it has been extensively used in designing schemas for database systems [26] and to represent the structures of systems in systems analysis [16]. An ER diagram not only provides an intuitive view of an application, but it is also possible to generate a relational schema automatically from it [84]. The ER approach has been very effective in the areas of data management and systems analysis.

2.2.1 Entity-Relationship Model

I now explain some key concepts used by the entity-relationship model. The entity-relationship model adopts a view that an application world consists of *entities* and *relationships* among them [15, 24].

Entity An *entity* is a cohesive unit of data [15]. An example of an entity is a person, company, or event.

Relationship A *relationship* is an association among entities. For instance, two person may be related to each other by a relationship `father-son` [15, 16].

Cardinality Ratio The *cardinality ratio* of each binary relationship type is a pair of numbers, each indicating the number of entities of one type associated with one entity of the other type. A cardinality ratio can be *one-to-one*, *one-to-many*, *many-to-one*, or *many-to-many*.

Properties or Attributes All entities of a given type have certain kinds of properties or attributes in common [24]. For example, every employee has an employee number, a name, a salary, and so on. Each property draws its value from the value set associated with it.

Subtype An entity type can be subtype of another entity type. For example, if some employees are programmers, then entity type `Programmer` is a subtype of entity type `Employee`. `Programmer` inherit all the properties of `Employee`.

2.2.2 Entity-Relationship Diagrams

An ER diagram represents the logical structure of a database in a pictorial manner [24]. I now explain notations used by ER diagrams as shown in Fig. 2.1 [15].

Entity Each entity type is shown as a rectangle labeled with the name of the entity type.

Relationship Each relationship type is shown as a diamond with lines connected to the entity types participating in the relationship type.

Cardinality Ratio A small filled-circle at the end of a line means "many". If a line has no small filled-circle at each end, a cardinality is "one".

Properties Properties are shown as labels encircled in ellipses attached to entity types or relationship types.

Subtyping The relationship of Y being a subtype of X is represented by a line between Y and X with a small arc on it.

Let us consider a queueing system as shown in Fig. 2.2. A queueing system consists of generators, queueing, and processors. A job generator produces a stream of jobs. A processor processes jobs, and a queue holds the jobs. The queueing system in Fig. 2.2 shows a simple queueing system consisting of one generator, three queues, and two processors. The ER diagram for queueing systems is shown in Fig. 2.3.

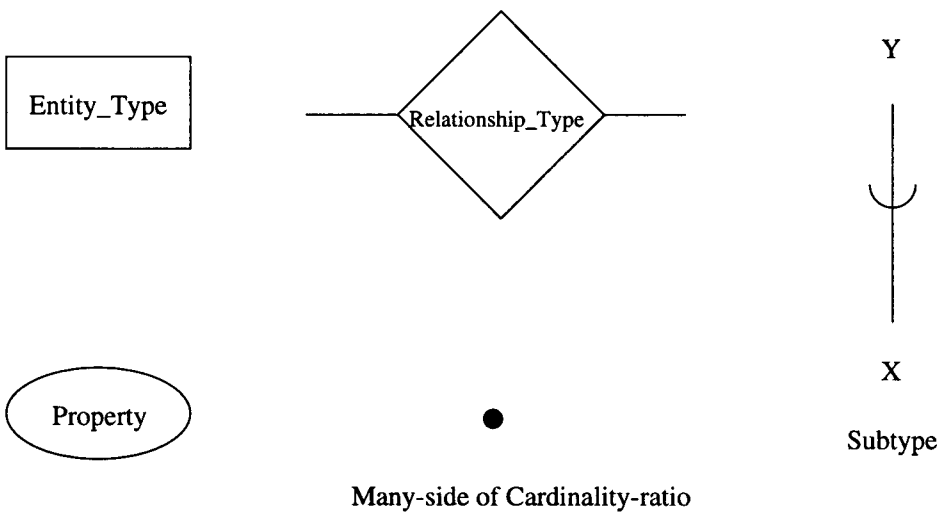

FIGURE 2.1: Entity-relationship notations.

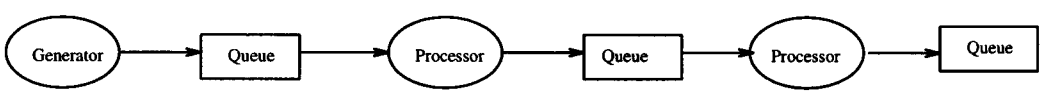

FIGURE 2.2: Simple queueing system.

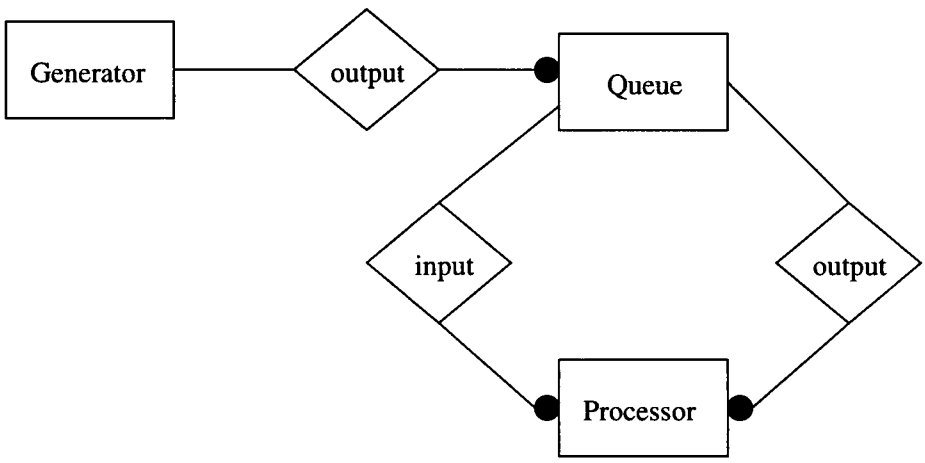

FIGURE 2.3: Entity-relationship diagram for queueing systems.

2.3 Object-Oriented Approach and Modeling

In the object-oriented (OO) approach, a system is conceived as a collection of *interacting objects*. Each object encapsulates both *state* and *behavior*. A *class* is used to define the structure of data and behavior of similar objects. A *class hierarchy* represents *superclass-subclass* relationships. A subclass inherits all the properties from its superclasses. The OO approach thus promotes software reusability.

An object-oriented analysis and design (OOD) methodology provides a set of rules that guides a programmer to analyze and design a system in terms of objects, classes and interactions among objects. An OOAD methodology makes easy a transition from design to implementation. Some well known object-oriented design methodologies are *Responsibility-Driven Design* [89, 8], *Object Modeling Technique* [72, 59, 70], the *Booch* method [12, 28, 19], and the *Fusion* method [17].

2.3.1 Responsibility-Driven Design

In Responsibility-Driven Design proposed by Beck and Cunningham [8], an application is modeled in terms of classes, responsibilities, and collaborations. Responsibilities of a class are categories of actions to be performed by the objects belonging to that class. In fulfilling its responsibilities, a class needs to collaborate with other classes. A CRC (Class, Responsibilities, and Collaboration) card is provided for each class to describe its responsibilities and collaborating classes.

2.3.2 Object Modeling Technique (OMT)

In OMT, an application is modeled by an *object model*, a *dynamic model*, and a *functional model* [72].

An object model describes the static structure of a system with two types of diagrams: *class diagrams* and *instance diagrams*. A class diagram is a schema, pattern, or template for objects instances. An instance diagram describes how a particular set of objects are related to each other.

A dynamic model describes the behavior of objects. The major dynamic modeling concepts are events and states. OMT uses *state diagrams* as a dynamic model . A state diagram is a graph whose nodes are states and whose arcs are state transitions caused by events.

A functional model describes computations within a system. The functional model shows how output values are derived from input values. The functional model consists of *data flow diagrams* (DFD), each of which is a graph showing the flow of data values and the transformations on them.

2.3.3 The Booch Method

The Booch method supports *class diagrams*, *object diagrams*, *module diagrams*, *state transition diagrams*, *interaction diagrams*, and *process diagrams* [11, 12, 87]. Class diagrams, object diagrams, and state transition diagrams used by the Booch method are similar to those usde by other methodologies.

A module diagram shows the allocation of classes and objects to modules. An interaction diagram shows a trace of messages generated by an execution of a scenario. An interaction diagram is equivalent to an event trace diagram of OMT [72] and is also used by the system of Ivar Jacobson [42]. A process diagram is used to show the allocation of processes to processors in the physical design of a system.

2.3.4 The Fusion Method

The fusion method supports an *object model*, an *operation model*, *object interaction graphs*, *visibility graphs*, and *class descriptions* [17].

An object model is similar to that used by other methodologies. An operation model describes the behavior of a system by specifying the effects of system operations in terms of events and state changes.

An object interaction graph shows the sequence of messages that occur when an object performs a particular operation. A visibility graph identifies for each class C whose instances are accessed by the instances of the class C. A directed arrow is used to show the possibility of access. A class description specifies for each class the superclass, the data attributes, the object reference attributes, and the method signatures.

An object model and an operation model are used in the analysis stage of a software life cycle, but an object interaction graph, a visibility graph, and a class description are used in the design stage.

2.4 Software Component Composition

Software component composition is a process of creating application software from their components by connecting them together. A component is a unit of coherent functionality. Components can be functions, procedures, modules, classes, objects, specifications, and documents [62, 63, 85].

In the following subsections I summarize various research on software component composition including software ICs, component engineering, gluons, visual composition, software architectures, active cookbooks, component software, JavaBeans, and ActiveX. None of these approaches uses an entity-relationship diagram as a template for constructing application software.

2.4.1 Software ICs

Brad J. Cox introduced the notion of Software ICs [20]. Software-ICs are binary files just like those in a conventional software library. Every Software-IC contains its own factory object, and it is used to produce objects supported by the Software-IC. Each Software-IC is described in specification sheets that provide the highly compressed technical information.

2.4.2 Component Engineering

Oscar Nierstrasz and Laurent Dami define software composition as the process of constructing applications by interconnecting software components through their plugs [63]. A composition mechanism may use *functional composition* or *object-oriented composition*. *Component engineering* should produce a software component framework that can satisfy different sets of requirements, even unknown requirements. The clients of an application are the end-users, whereas the clients of a component framework are the application developers.

2.4.3 Gluons

Xavier Pintado presented *gluons* as objects that mediate cooperation among software components [65, 66]. Whenever a gluon receives a message it forwards the message to the object that has been designated as its server.

2.4.4 Visual Composition

Vicky de May defines *visual composition* as the interactive construction of running applications by direct manipulation of visually-presented software components [25]. *A composition model* describes as texts the set of rules for composing components in a particular domain. Components communicate with each other via links and ports. The connections between components are guided by the plug-compatibility rules specified within the composition model. Vista is a prototype environment based on this approach [25].

2.4.5 Structural Active-Object System (SAOS)

A structural active-object system (SAOS) program is constructed as a collection of *structurally* and *hierarchically* composed active objects [56, 58, 57, 55]. The behaviors of these active objects are defined by *transition statements*, which are *transition rules*, `always` *statements*, *future calls*, or *future assignments*. SAOS programs were implemented first in C++ and then in Java [53].

Each transition rule is a condition-action pair, whose action part is executed when its condition part is satisfied. An `always` statement is an equational assignment statement that maintains an invariant relationship (constraint) among the states of objects. *Future calls* and *future assignments* are used to activate future events.

In most cases, behaviors can be best described by transition rules. `always` statements are useful to update the attributes of graphical objects automatically. Future

calls and assignments are convenient to schedule delayed actions, although they are used less frequently compared to transition rules or `always` statements.

2.4.6 Software Architecture and Design Patterns

Mary Shaw, *et al.* describe the architecture of a software system in terms of the components used, the interactions among those components, and the patterns that guide the composition of components into systems [5, 35, 32, 34, 33, 78, 79, 81]. An architectural style is defined as a collection of recurring patterns of system organizations. In other words, it is an abstract framework for related applications. Some examples of architectural styles are procedure-call, pipe-filter, pipeline, realtime, and event-based styles. In their research, a variety of components and their connections are identified and classified. They implemented the architecture description language *Unicon*: Language for Universal Connector Support. The focus of the language is to support a variety of architectural parts and styles found in real applications.

Gamma, *et al.* also discuss design patterns [31]. Design patterns promote reuse of designs and architectures.

2.4.7 Active Cookbooks

Albert Schapert, *et al.* divided a development process into two activities: creation of a framework providing new components and composition of applications by using components supported by the framework [75]. They used the *relations* among the software components as a basis for abstraction, reuse, and automatic code generation. They also proposed an *active cookbook* which consists of online recipes that guide software developers in the use of framework.

2.4.8 Component-Based Software

Component-based software allows the assembly of independently developed components. This approach addresses the general problem of designing an application from software components that were constructed independently by different developers with different languages and tools [6].

2.4.8.1 OLE 2.0

OLE 2.0 (Object Linking and Embedding) of Microsoft is a set of standard specifications and implementations for component-based software [13]. OLE 2.0 is based on Component Object Model (COM), which ensures binary-level interoperability across application components written in different languages.

2.4.8.2 JavaBeans

JavaBeans allows component-based software development in Java. Each bean, which is a Java `Component` that interact with one another [3], possesses *introspection*, *properties*, and *event-handling capability.* The introspection enables an application builder tool to analyze how a bean works, what methods it supports, and what its states are. The event-handling capabilities of beans allow them to be connected and to communicate with each other. The properties enable developers to customize a bean. A BeanBox is a sample `Container` for testing beans [4].

One goal of the JavaBeans architecture is to provide a platform neutral component architecture. For example, the JavaBeans APIs can be linked to COM and ActiveX on Microsoft platforms.

2.4.8.3 ActiveX

ActiveX, which is based on Component Object Model (COM), is a set of technologies that enable developers to combine the outputs of many different programming lan-

guages into a single, integrated Web page [2]. This gives Web designers the flexibility to include in a page multiple objects programmed in different languages.

ActiveX controls are graphical-user interface objects that can be embedded in a Web page. Web developers can create an interactive Web page by using reusable controls available in the market. ActiveX Documents, an ActiveX facility, enables users to view non-HTML documents such as Microsoft Excel or Word files through a Web browser. ActiveX scripts control the behaviors of ActiveX controls and Java applets from the browser or server.

2.5 Software Development Environments

Building an application with a software development environment is one promising idea. A software development environment aims at automating some activities in a software development life-cycle. Dart *et al.* classified software development environments into *language*-based, *structured*-based, *toolkit*-based, and *method*-based ones [23]. Biggerstaff *et al.* categorized software development environments into *generation*-based and *component*-based ones [9]. Fuggetta defined five classes of CASE environments: toolkits, languages-centered environments, integrated environments, fourth generation environments, and process-centered environments [30].

In this section I describe three types of visual software development environments: *programming*-based, *generation*-based, and *component*-based environments.

2.5.1 Programming-Based Environments

A programming-based environment is used to support the coding phase of a software development cycle. It mostly supports editing, compiling, and debugging activities in programming-in-the-small. Some environments are designed for general languages and others for existing languages. Examples of environments for special languages are

PROGRAPH [21] and PECAN [71]. Examples of environments for existing languages are Interlisp for Lisp language [83], Cedar for Mesa/Cedar [82], Smalltalk/V for Smalltalk [36], and the Relational Environment for Ada [7].

2.5.2 Generation-Based Environments

A generation-based environment reuses patterns integrated into a generator program [9]. Reusable patterns may be code patterns or rule patterns used by transformation systems. An example of a generation-based environment is the Draco system, which is an application generator and transformation-based system [61].

2.5.3 Component-Based Environments

Component-based environments allow a user to build up application from software parts or components. Software parts can be code skeleton, subroutines, functions, programs, and objects. Examples of component-based environments are HyperCard [38], ACE [43], ITS [88], CONDOR [45], REX [47], AVS [1], HP VEE [40], and NUT [86].

HyperCard is an authoring tool and information organizer based on the concept of a stack of information. ACE and ITS are used to build interactive graphical applications. CONDOR is an interactive constraint-based dataflow programming environment for computer graphics. CONDOR has a graphical user interface in which mathematical functions are represented as boxes with vector or scalar inputs and outputs. REX represents a distributed and parallel system as a set of an interconnected component instances.

Application Visualization System (AVS), an interactive visualization environment, allows the user to interactively build a network of modules that handle data input, filtering, mapping, and rendering. Hewlette-Packard Visual Engineering Environment (HP VEE) is a graphical programming environment that allows an ap-

plication to be constructed from interconnected icons. NUT is a knowledge-based programming environment consisting of a window-based interactive user interface, a language processor, and a graphics facility. The NUT graphical tools include a graphical editor, run-time graphical functions, and a scheme editor.

Chapter 3

ENTITY-RELATIONSHIP SOFTWARE DEVELOPMENT ENVIRONMENT (ERSDE)

In this section I present the architecture of Entity-Relationship Software Development Environment (ERSDE). As shown in Fig. 3.1, the ERSDE consists of three major parts: the *entity-type editor*, the *schema editor*, and the *application editor*. The major feature of the ERSDE is that it can be used to develop executable application software in different application domains by component composition.

The meta schema, which conceptually consists of the *entity metatype* and the *relationship metatypes* as shown in Fig. 3.2, is a template for creating *domain-specific schemas*. The entity-type editor created iconic entity types by instantiating the entity metatype. 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 section 3.2.

A domain-specific (application-specific) schema is an EERD consisting of entity types and relationship types among them. An example of an EERD for tank systems is shown in Fig. 3.3. The schema editor is used to construct a domain-specific schema from the entity types created by the entity-type editor and by instantiating the relationship metatypes. 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 its menu. The application editor allows applications to be composed

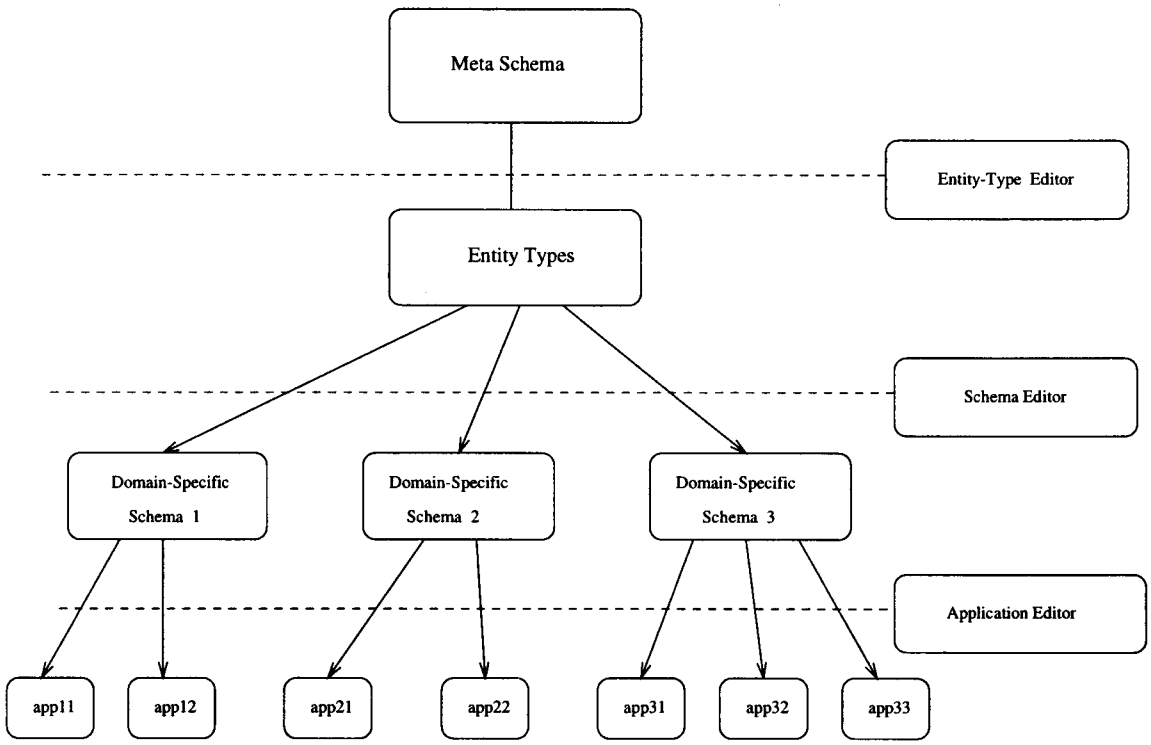

FIGURE 3.1: Structure of the ERSDE software development environment.

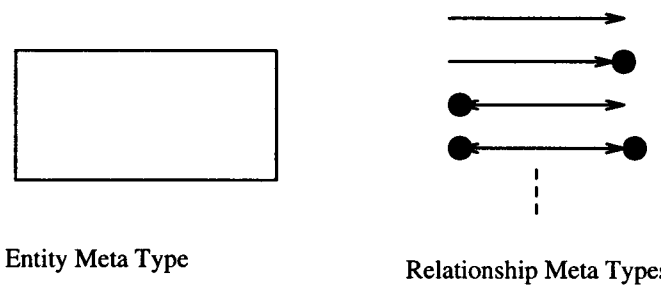

FIGURE 3.2: Simplified meta schema.

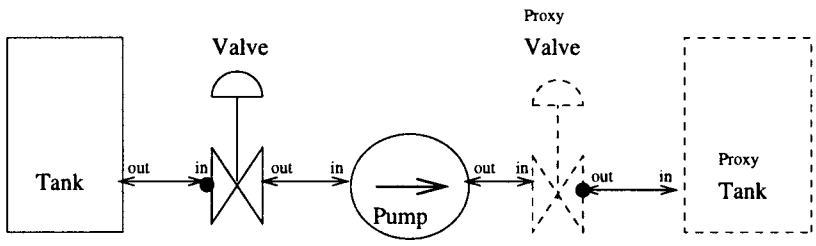

FIGURE 3.3: Extended entity-relationship diagram with proxy of entity types.

in different application domains by switching the EERD used as the menu. In the next three sections, I describe more details of the three major parts of the ERSDE.

3.1 Entity-Type Editor

The entity-type editor, as shown in Fig. 3.4, is used to create entity types to be used in EERDs. The entity metatype provides the rules to be used in creating entity types. 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 entity type to inherit some characteristics from other entity types. When we create an entity type with entity-type editor, we can specify its supertype.

3.2 Schema Editor

We use the schema editor to build domain-specific schemas. A domain-specific schema displays the entity types and the relationship types used by the application in that domain. The schema editor, as shown in Fig. 3.5, is a (general) domain-independent graphical editor for creating and manipulating graphical representations of entity types and relationship types in domain-specific schemas.

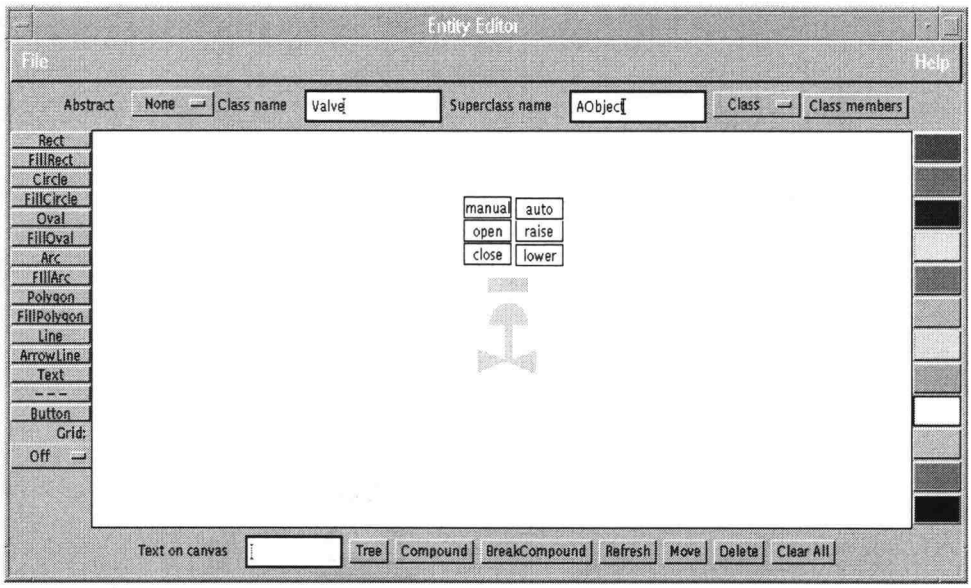

FIGURE 3.4: Entity-type editor.

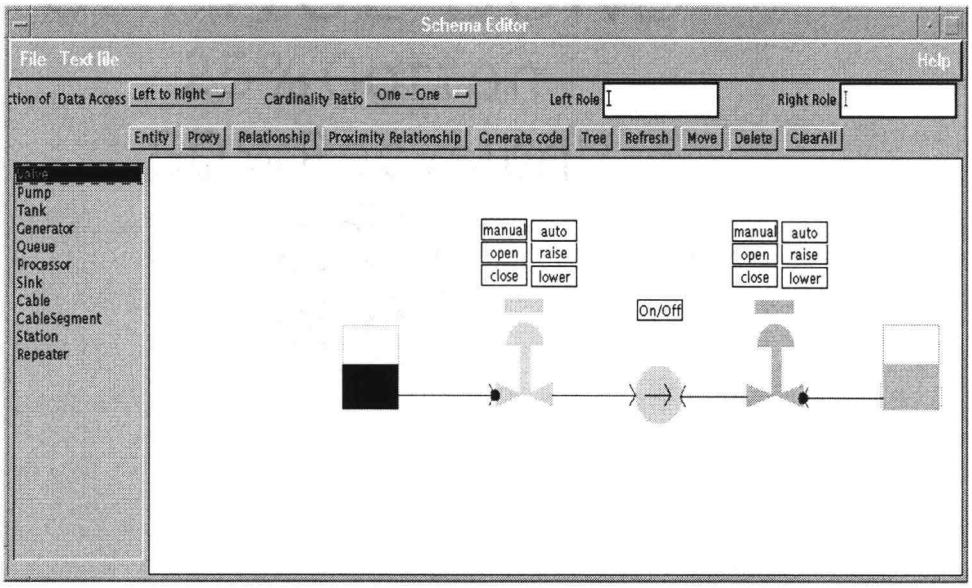

FIGURE 3.5: Schema editor.

The meta schema is a template for creating domain-specific schemas. I adopt some conventional notations and propose some new ones for the meta schema. The meta schema as shown in Fig. 3.6 provides notations for the *entity composition*, *relationship types*, and *proxy entity-types*. The first two notations are used extensively in many object-oriented design methods including OMT, Booch, Fusion [72, 12, 17]. However, in this 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 I do. However, only one way of data referencing is allowed [17]. The concepts of proxy entity types and relationship representation by proximity, which I describe later, are new.

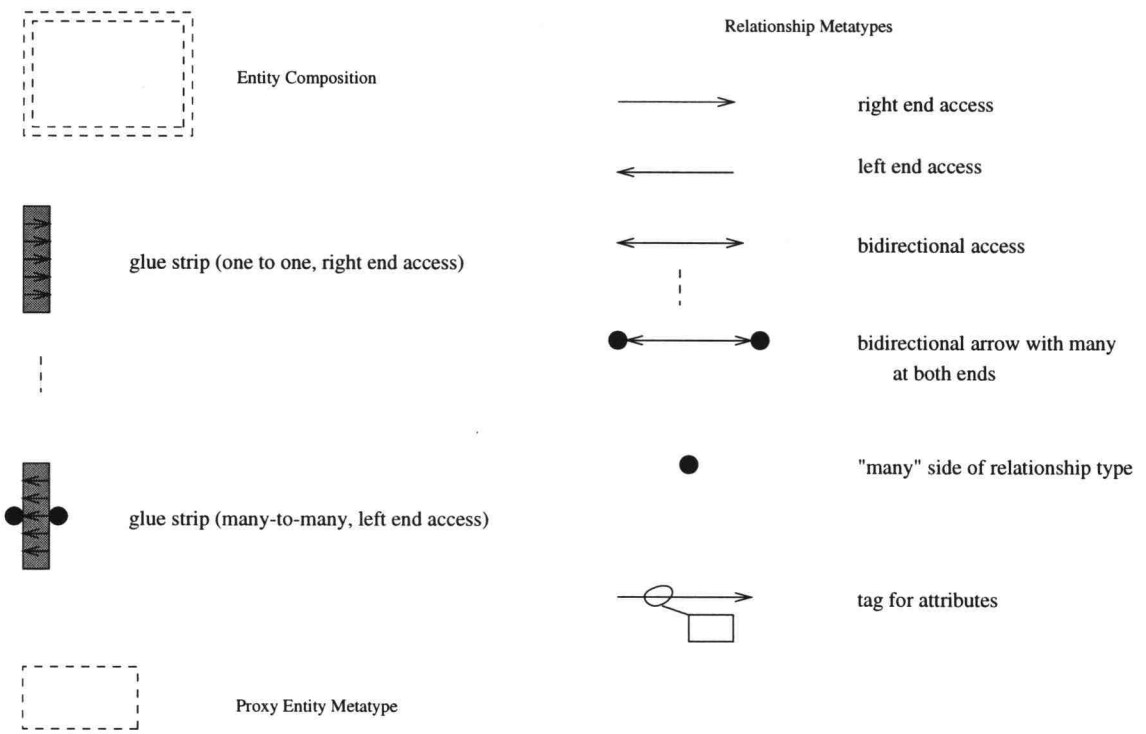

FIGURE 3.6: Meta schema.

3.2.1 Entity Composition

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.

3.2.2 Relationship Types

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. I now describe cardinality ratios, direction of data access, and roles of relationship types.

3.2.2.1 Cardinality ratios

A relationship metatype is a *unidirectional* or *bidirectional* arrow with small filled-circles at its ends. A small filled-circle (•) 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 thesis, 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.

3.2.2.2 Direction of Data Access

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): An instance of E2 can access an instance of E1.

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. 3.7. Fig. 3.7(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. 3.7(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. 3.7(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. 3.7(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. 3.7(a), Fig. 3.7(b), and Fig. 3.7(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. 3.7(c), Fig. 3.7(f), Fig. 3.7(j) are examples of one-to-many relationship types whose attributes can be moved to entity type E2. Fig. 3.7(d) Fig. 3.7(e), Fig. 3.7(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. 3.7(g), Fig. 3.7(h) and Fig. 3.7(l) are examples of many-to-many relationship types.

3.2.2.3 Roles

A role is one end of an association [72]. Each entity participating in relationship performs a particular role in that relationship. Such a role can be designated at the schema level by assigning a role name to each end of a relationship type. Fig. 3.7 shows relationship metatypes with roles.

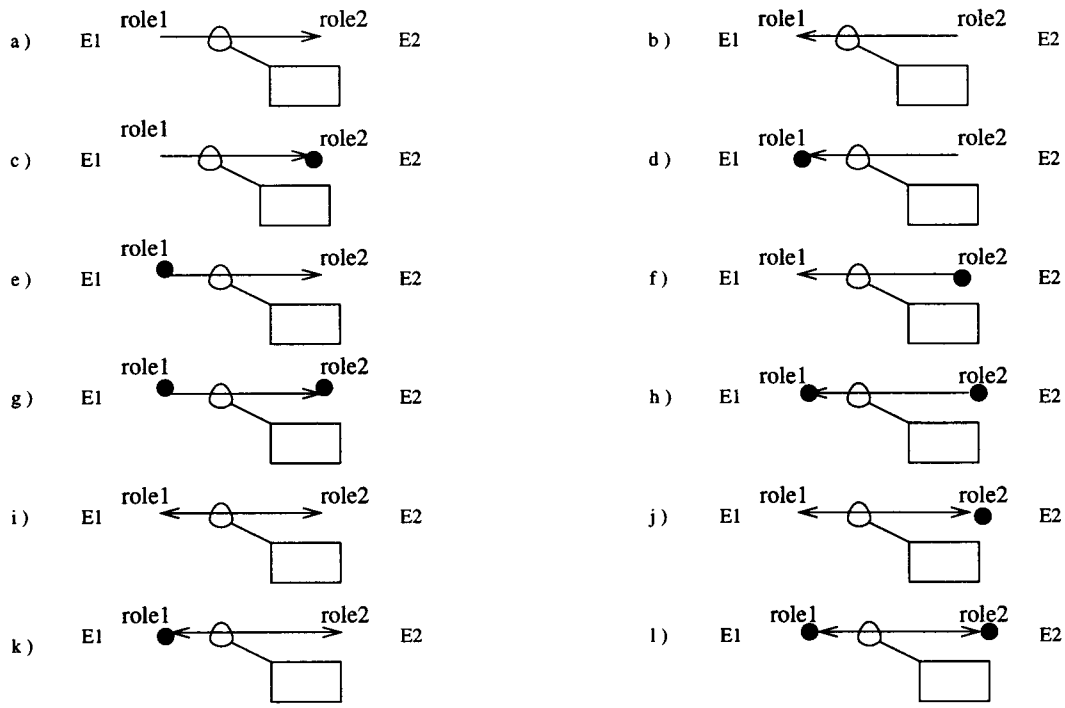

FIGURE 3.7: Relationship metatypes between two entity types.

3.2.3 Proxy Entity-Types

I 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.

3. (Multiple Component Problem) This problem occurs when a composite entity type includes multiple occurrences of one entity type as its components. Fig. 3.8 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. 3.9, on the other hand, shows the composite entity type `Car` by using proxy entity types.

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.

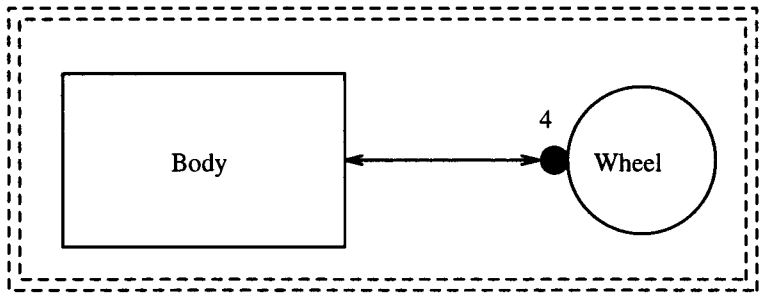

FIGURE 3.8: A car as a composite entity type.

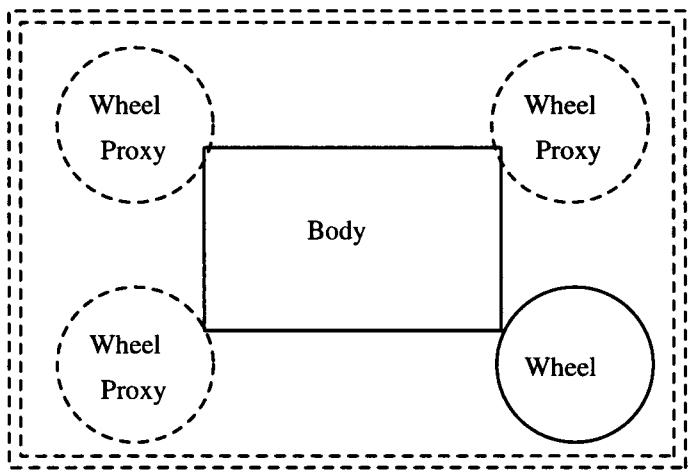

FIGURE 3.9: A car as a composite entity type with proxies.

3.2.4 Creating a Domain-Specific Schema or an EERD

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.

I 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. 3.10. A `Car` has one `Driver` and multiple `Passengers`.

3.3 Application Editor

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

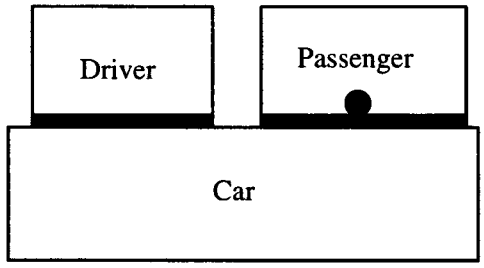

FIGURE 3.10: The EERD for a car, a driver and passengers.

an EERD is more effective than a conventional editor menu since it can show not only entity types but also possible relationships between entities.

The application editor, which is shown in Fig. 3.11, is a general graphical editor used to create, and move instances of entity types and connections among them. If each entity is an active object, the application can be executed once the entities are connected. The application editor allows us to construct applications in different layouts, and to move, delete, and edit components interactively.

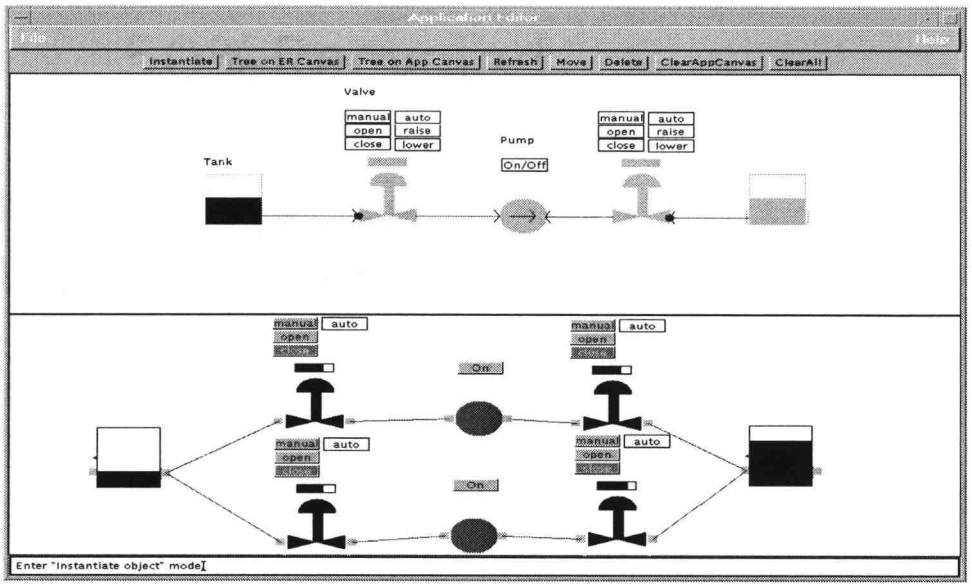

FIGURE 3.11: Application editor.

Chapter 4

USING THE ENTITY-RELATIONSHIP SOFTWARE DEVELOPMENT ENVIRONMENT (ERSDE)

We have described the architecture of the *Entity-Relationship Software Development Environment* (ERSDE) in the previous chapter. We now explain how to use the ERSDE focusing primarily on its three editors: the *entity-type editor*, the *schema editor*, and the *application editor*.

4.1 Entity-Type Editor

An entity type, which uses an iconic representation to model an entity, can be created with the entity-type editor. A (primitive) entity type is represented either by one graphical object or by a *compound* graphical object. A "composite" entity type discussed in section 4.2 is different from a "compound graphical object". A primitive entity type may be represented by a compound graphical object. A composite entity type consists of multiple entity types. Fig. 4.1 shows the entity-type editor screen with the iconic representation of entity type `Generator` along with two dialog windows. The entity-type editor consists of

1. a drawing canvas in the middle,

2. a menu of buttons for graphical operations in the bottom row,

3. a menu of graphical objects in the leftmost column,

4. a color palette in the rightmost column,

5. various fields and buttons in the third row from the top, and

6. the file and help menus in the second row from the top.

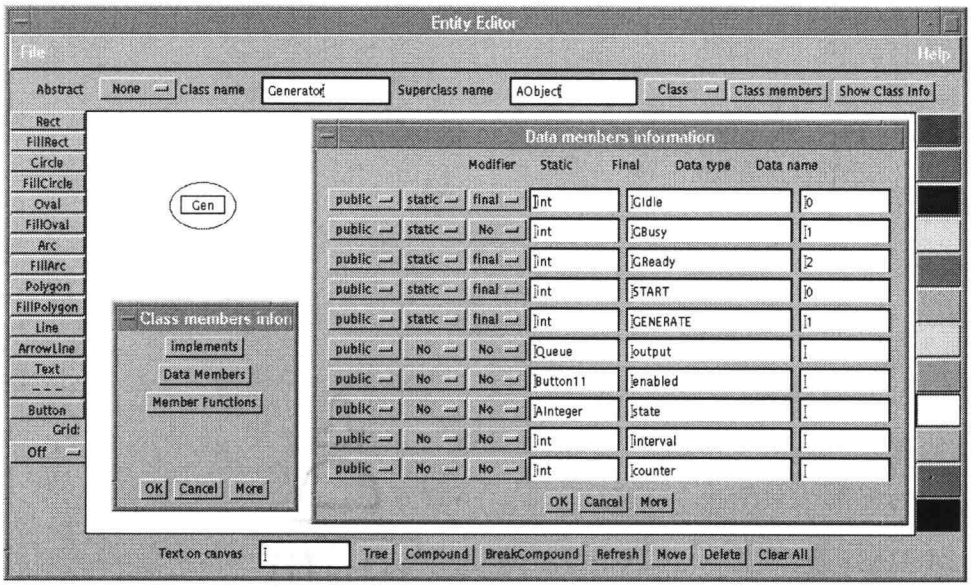

FIGURE 4.1: An entity editor screen with primitive entity type `Generator` on canvas.

4.1.1 Creating Graphical Objects

We can create a graphical object by using the entity-type editor like a drawing program. A color for a graphical object to be drawn can be selected from the color palette. Rubberbanding is used in creating such graphical objects as rectangles and circles.

4.1.2 Specifying Data Members for an Entity Type

A user can specify the data members for each entity type with the commands `Class Members` and `Data Members`. When the command button `Data Members` is selected, the data-member information dialog with ten entries is displayed. For each entry for a data member the user can select three modifiers and provide texts in three fields. The three modifiers must be selected between `public` and `private`, between `static` and `not-static`, and between `final` and `not-final`. A data type, a data name, and a data value can be provided as text for each data member.

If the current entity type has more than ten data members, the user can click on the command button `More` located at the bottom of the dialog to have the next ten entries to be displayed.

4.1.3 Specifying Member Functions for an Entity Type

A user can specify the member functions for each entity type with the command button `Class Members` and the command button `Member Functions`. When the command button `Member Functions` is selected, the member-function information dialog with ten entries is displayed. For each entry the user can select three modifiers, provide texts in two fields, and select the command button `Arguments` if needed. Selections of the modifiers are between `public` and `private`, between `static` and `not-static`, and between `final` and `not-final`. The return type and the function name can be specified as texts for each member function.

The argument information dialog is displayed when the command button `Arguments` is pressed. This dialog has ten pairs of text fields. The argument type and the argument name can be specified as texts for each argument. If the entity type has more than ten member functions, the user can click on the command button

`More` which is located at the bottom of the member function dialog. Then the next ten entries will be displayed.

4.2 Schema Editor

We can create an EERD with the schema editor by placing entity types and defining relationship types among them. Fig. 4.2 shows the schema editor screen with the EERD for queueing systems on its canvas. The schema editor consists of

1. a canvas in the middle,
2. a menu of buttons for editing operations in the fourth row from the top,
3. a menu of entity types in the leftmost column,
4. choices for a relationship type in the third row from the top, and
5. the file and help menus in the second row from the top.

4.2.1 Placing Entity Types and Creating Proxy Entity Types

To place an entity type on the schema editor canvas, perform the following steps.

1. Enter the *entity-type-placement* mode by clicking the left mouse button with the cursor on the command button `Entity`.
2. Select an entity type from the entity-type menu.
3. Click the left mouse button at any location inside the schema editor canvas. Then the icon of the selected entity type will be drawn there.
4. By repeating step 3 entity types created after the first one becomes proxy entity-types.

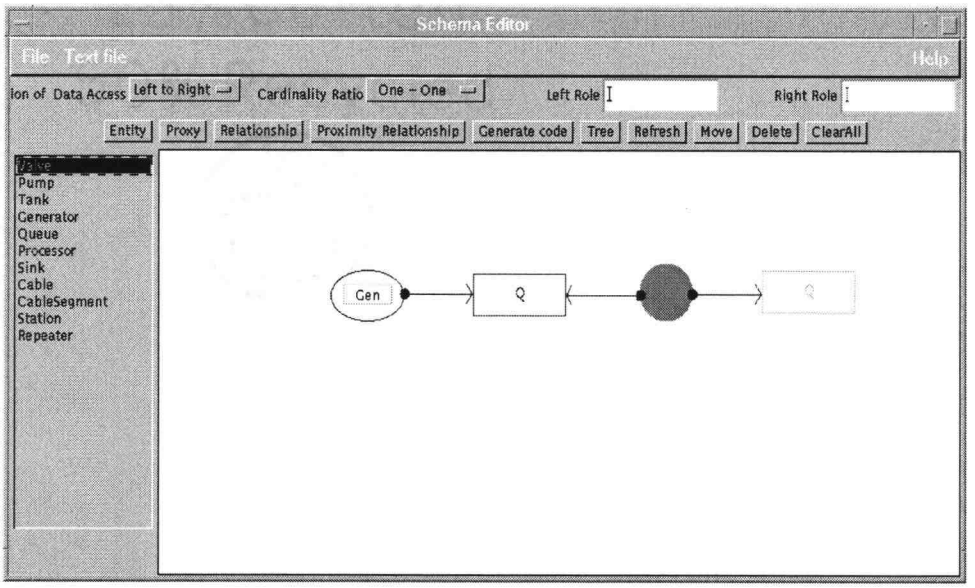

FIGURE 4.2: The schema editor with the EERD for queueing systems on canvas.

4.2.2 Creating Relationship Types by Links

A relationship type is represented as a link between two entity types. To create this link, perform the following steps.

1. Enter the *relationship-type-creation* mode by clicking the left mouse button with the cursor on the command button `Relationship`.

2. Select a direction of data access, which may be `left-to-right`, `right-to-left`, or `bidirectional`.

3. Select a cardinality ratio, which may be `one-one`, `one-many`, `many-one`, or `many-many`.

4. Entity participating in association may have role. At each end of an association we assign a role name. Fill in the two textfields for the roles of the entity types on the left and right side of the relationship type.

5. Drag the cursor on the canvas while pressing the left mouse button, starting inside some entity type and ending inside another entity type. An arrow representing the relationship type between these two entity types is created when the left mouse button is released.

4.2.3 Creating Relationship Types by Proximity

A *relationship type by proximity* between two adjacent entity types is represented by a glue strip. To create this glue strip, perform the following steps.

1. Enter the *proximity-relationship-type-creation* mode by clicking the left mouse button with the cursor on the command button `Proximity Relationship`.

2. Select a direction of data access, which may be `left-to-right`, `right-to-left`, or `bidirectional`.

3. Select a cardinality ratio, which may be `one-one`, `one-many`, `many-one`, or `many-many`.

4. Fill in the two textfields for the roles of the entity types on the left and right side of the relationship type.

5. Create the glue strip, which is represented by a rectangle strip between two adjacent entity types, by dragging mouse with the left mouse button pressed until the right size of the glue strip is reached. A small rectangle represent-

ing the proximity relationship type is created when the left mouse button is released.

4.2.4 Creating Composite Entity Types

There are two kinds of composite entities: a *record* of entities and an *array* of entities.

4.2.4.1 Record Entity Type

A record entity type is represented by multiple entity types and surrounded by a double dashed-line roundtangle.

4.2.4.2 Array Entity Type

An array entity type is represented as following. Two copies of one entity type are placed apart horizontally or vertically, and three-dots are placed between these two entity types. The array entity type is surrounded by a double dashed-line roundtangle. Fig. 4.3 shows a schema editor screen with an array composite entity type `Cable` in it.

4.2.5 Code Generation

Once the EERD is completed, skeleton code can be generated by clicking the command button labeled "`Generate Code`". This skeleton code contains a collection of classes, their properties, and the pointer structures for relationship types. The exact pointer structures generated are determined by the directions of data access and the cardinality ratios of the relationship types.

In the remainder of this subsection, we discuss the skeleton code generated for application-specific classes. We then show the code that must be added by the user for behavior descriptions. Finally, we explain the code used to connect entities together.

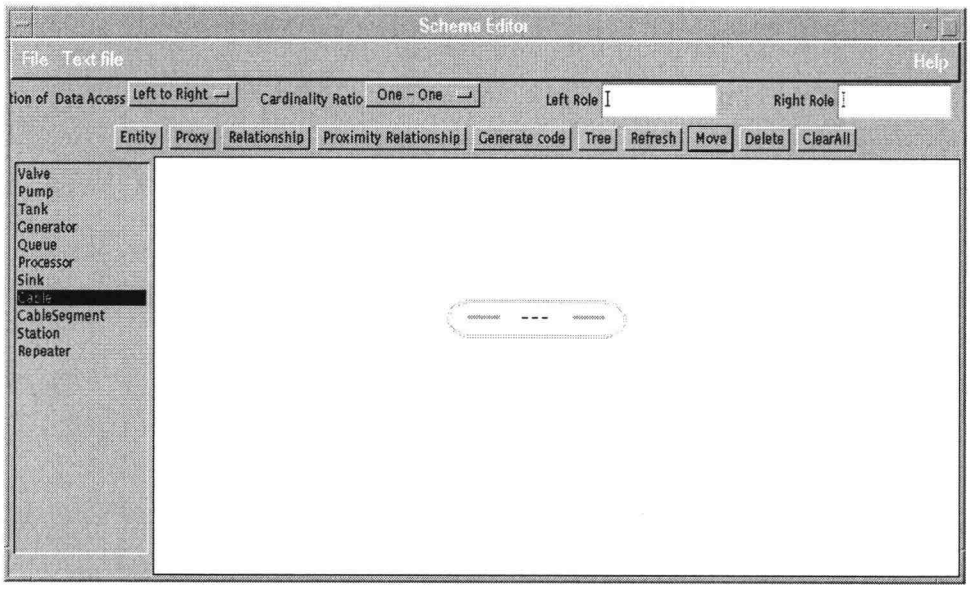

FIGURE 4.3: An schema editor screen with an array entity type `Cable` in it.

4.2.5.1 Skeleton Code Generated

We first explain the skeleton code generated for each entity type in the EERD. For this purpose, we use, as an example, some of the classes used by queueing systems: classes `Processor`, `EditProcessor`, and `QueueSystem`. Other classes generated for this application are shown in Appendix A.

Two classes are generated for each entity type in the current EERD: the class for the model and the class for the view. The model class is a subclass of `AObject`. A model object maintains the data and behavior without a graphical representation. The view class is a subclass of `EditApp`. The view object provides a graphical representation. The name of a view class begins with "`Edit`" and is followed by the name of the entity type, e.g., "`EditProcessor`".

Class `Processor` shown in Fig. 4.4, is a model class and Class `Editprocessor` shown in Fig. 4.5 is its view class. There must be a link from the view to the model.

For example, the member variable `model` of view class `EditProcessor` shown in Fig. 4.5 points to an instance of `Processor`. The class `Processor` generated contains the data members and the member functions that provided with the entity-type editor.

The pointer structures `input` and `output` are automatically generated from the relationship types associated with entity type `Processor`. For class `Processor`, the schema editor generates data members `model`, `input`, and `output` for connections, and the method `initialize()`. The constructor performs the following tasks:

1. creates an instance of the model class `Processor` and sets it to `model`,
2. creates the view object by cloning the graphical object in the EERD, and
3. instantiates `PortView` for `inPortView` and `outPortView`.

The method `initialize()` performs the following tasks:

1. inserts the `Processor` assigned to `model` into the Active Object System,
2. adjusts the coordinates of the view object, and
3. initializes the `PortViews` assigned to `inPortView` and `outPortView`.

Class `QueueSystem` shown in Fig. 4.6 is the top-level application class that handles entity instantiations. Each entity is instantiated by invoking the method

```
createObject(index, entity_type, x, y).
```

Argument `index` indicates the entity type whose instance should be instantiated. Argument `entity_type` points to the graphical object representing the selected entity type in the EERD. A view object is cloned from the graphical object of the selected

```
public class Processor extends AObject
{
  public static final int PIdle = 0;
  public static final int PBusy = 1;
  public static final int PComplete = 2;
  public static final int START = 0;
  public static final int STOP = 1;
  public static Random rand = new Random();
  public AInteger state;
  public int maxProcessingTime = 6;
  public int counter;
  public Queue input;
  public Queue output;

  public Processor(String name)  {}  // to be filled by a user
  public void paint (Graphics g) {}
  public void start () {}
  public void stop () {}
  public void dispatch (int funcIndex) {}
  public void initialize () {}
  public void setInput(PortView port) {}
  public void setOutput(PortView port) {}
  public void print (int level) {}

}
```

FIGURE 4.4: Automatically-generated skeleton code for class `Processor`.

```
class  EditProcessor  extends  EditApp implements Constants
{
  Processor  model;
  PortView   inPortView;
  PortView   outPortView;

  public EditProcessor(EditCompound object,
                       AppCanvas canvas, int x, int y)
  {
    super(x, y);
    model = new Processor("Processor");
    view  = (EditCompound) object.clone();
    this.canvas = canvas;
    name = new String("Processor");
    inPortView = new PortView(IN, canvas, this, SINGLE);
    outPortView = new PortView(OUT, canvas, this, SINGLE);
  }
  public void initialize() throws SaosException
  {
    insert(model, true);
    int diffX = x - view.x;
    int diffY = y - view.y;
    view.updateCoordinates(x,y, diffX, diffY);
    view.setColor(Color.green);
    view.initialize(canvas.getGraphics());
    inPortView.initialize();
    outPortView.initialize();
   }
  public void setPtr(PortView port) throws SaosException
  {
    if (port.role == IN)
      model.setInput(port);
    if (port.role == OUT)
      model.setOutput(port);
  }
}
```

FIGURE 4.5: Automatically-generated skeleton code for class `EditProcessor`.

entity type and is placed on the canvas. Class `QueueSystem`, which is application-dependent, is used by the application editor to create application-specific objects. We discuss the interface `Application` in section 4.3.3.

4.2.5.2 Code for Behavior Descriptions

The code for behavior descriptions must be added manually to classes `Processor` and `EditProcessor`. Behavior descriptions are done by member methods. The member methods of class `Processor` are shown in Fig. 4.7, and those of class `EditProcessor` are given in Fig. 4.8. The completed code for other classes is shown in Appendix B.

The five methods of `Processor` are `initialize()`, `start()`, `stop()`, `setInput()`, and `setOutput()`. The method `initialize()` associates with the methods `start()` and `stop()` the active variable `state`. The method `start()` initiates processing a job, decrements the number of jobs in the `Queue` designated by `input` by one, and schedules a future assignment for active variable `state`. The method `stop()` deposits the processed job to the `Queue` designated by `output` and increments the number of the jobs in it by one. We explain the methods `setInput()` and `setOutput()` in the next subsubsection.

The three methods of `EditProcessor` are `initialize()`, `ifavail()` and `setPtr()`. The method `initialize()` associates the method `ifavail()` with the active variable `state`. The method `ifavail()` is used to change the color of the view of the `EditProcessor` when the state of the `Processor` designated by `model` changes. The method `setPtr()` connects the `Processor` to a `Queue` as specified by the relationship type in the EERD. Further details on this method is discussed in the next subsubsection.

```
class QueueSystem extends AObject implements Application
{
  public EditApp createObject(int index, EditObject object,
                              int x, int y)
  {
    switch (index)
    {
      case 0:                    // create a Generator
        EditGenerator editGenerator =
              new EditGenerator( (EditCompound)
                                  (object), canvas, x, y);
        insert(editGenerator, true);
        return editGenerator;
      case 1:                    // create a Queue
        EditQueue editQueue =
              new EditQueue( (EditCompound)
                             (object), canvas, x, y);
        insert(editQueue, true);
        return editQueue;
      case 2:                    // create a Processor
        EditProcessor editProcessor =
              new EditProcessor( (EditCompound)
                                  (object), canvas, x, y);
        insert(editProcessor, true);
        return editProcessor;
      case 3:                    // create a Queue
        EditQueue editQueue1 =
              new EditQueue( (EditCompound)
                             (object), canvas, x, y);
        insert(editQueue1, true);
        return editQueue1;
      default: return null;
    }
}
```

FIGURE 4.6: Automatically-generated skeleton code for class `QueueSystem`.

```
public class Processor extends AObject
{ public void initialize() throws SaosException
  { state.addTE((AObject) this, START, "start()", SaosMain.AosUser);
    state.addTE((AObject) this, STOP, "stop()", SaosMain.AosUser);
  }                 // start job processing
  public void start() throws SaosException
  {  // start processing if input has jobs and processor is free
    if (input.nJobs.val > 0 && state.val == PIdle)
    {                           //remove a job from input queue
      input.nJobs.decrement();
      counter++;
      state.setVal(PBusy);      // make processor busy
    // sets state to PComplete after processing completes
      FAssign.fAssign((AObject) this, state,
               PComplete, Math.abs(rand.nextInt()
               % maxProcessingTime), "state", SaosMain.AosUser);
    } }
  public void stop() throws SaosException
  {   if (output != null)
      if (state.val == PComplete && output.nJobs.val < output.nSlots)
      {
        output.nJobs.increment();
        if (SaosMain.debugLevel.val > 1)
        state.setVal(PIdle);
    } }
  public void setInput(PortView port) throws SaosException {
    input = (Queue) port.address.objectPtr;
    input.nJobs.addTE((AObject) this, START, "start()",
            SaosMain.AosUser);
  }
  public void setOutput(PortView port) throws SaosException {
    output = (Queue) port.address.objectPtr;
    output.nJobs.addTE((AObject) this, STOP, "stop()",
            SaosMain.AosUser);
 }
}
```

FIGURE 4.7: Behavior code for Processor.

```
class  EditProcessor  extends  EditApp implements Constants
{
  Processor model;
  public void initialize() throws SaosException
  {
    .
    .
    .
    try
    {
      model.state.addTE((AObject) this,
                  PIdle, "ifavail()", SaosMain.AosUser);
    } catch (SaosException er)
    {
      System.out.println("ERROR: " + er);
    }
  }

  public void ifavail()
  {
    if (model.state.val == PIdle)
      view.setColor(Color.green);
    else
      view.setColor(Color.red);
    view.paint(this.canvas.getGraphics());
  }
}
```

FIGURE 4.8: Behavior code for `EditProcessor`.

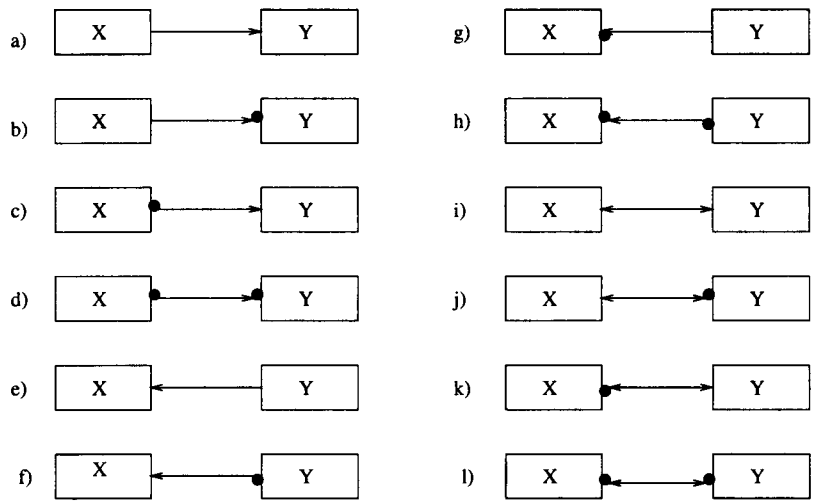

FIGURE 4.9: Relationship types between two entity types.

4.2.5.3 Code for Connecting Entities

In this subsubsection, we discuss the code generated by the schema editor for the pointer structures implementing relationship types. We first explain how the cardinality ratios and the directions of data access of the relationship types affect the code generated. There are three cases for data access, i.e., left-to-right ($\longrightarrow$), right-to-left ($\longleftarrow$), and bidirectional ($\longleftrightarrow$) and four cases for cardinality ratios, i.e., 1:1, 1:M, M:1, and M:M. Fig. 4.9 shows the twelve combinations of these cases.

The pointer structures implementing a relationship type between `X` and `Y` is determined as follows.

1. If the direction of data access is from `X` to `Y`, a pointer structure must be provided in `X`.

2. If the direction of data access is from `Y` to `X`, a pointer structure must be provided in `Y`.

3. If the direction of data access is bidirectional, a pointer structure must be provided in each of `X` and `Y`.

4. If the multiplicity of the cardinality ratio of the relationship is `1` at the end of the destination side of the data access, then a single pointer to the destination object is sufficient.

5. If this multiplicity is `Many`, then a vector of object references to the destination objects is needed.

The field names for the pointer structures implementing a relationship type is determined by the role names attached to the two ends of that relationship type. Fig. 4.10 shows role name `destination` or `destObjects` for the relationship type between entity types `X` and `Y`. The field name `destination` or `destObjects` in `X` as shown in Fig. 4.12 or Fig. 4.13 are obtained from the role name attached to the other end of the relationship type when seen from entity type `X`.

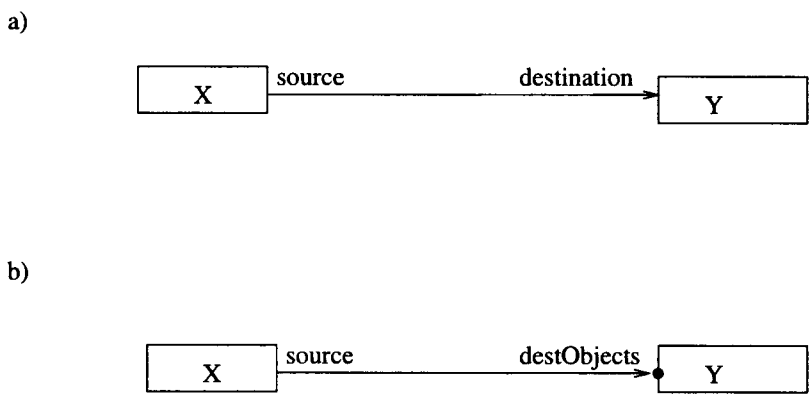

FIGURE 4.10: Role names associated with the relationship type between entity types `X` and `Y`.

In the cases of Fig. 4.9(a) and Fig. 4.9(i), an instance `x1` of `X` accesses at most one instance `y1` of `Y` as shown in Fig. 4.11(a). In the case of Fig. 4.9(c), multiple instances `x1, x2, and x3` of `X` access at most one instance `y1` of `Y` as shown in Fig. 4.11(c). For these cases shown the code as shown in Fig. 4.12 must be generated for class `X`. The field `destination` is a single pointer to an instance of Y.

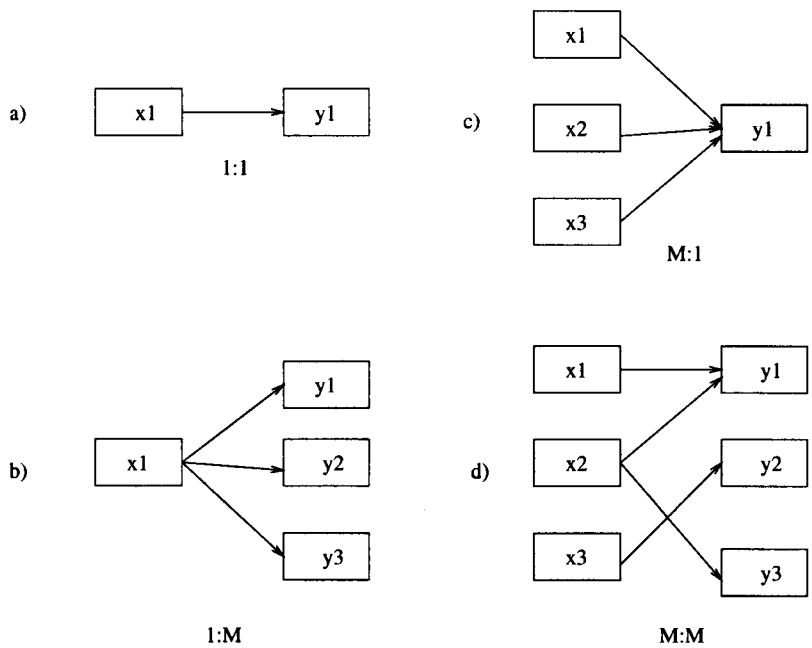

FIGURE 4.11: Relationships among entities of two entity types `X` and `Y`.

In the cases of Fig. 4.9.(b), Fig. 4.9.(d), and Fig. 4.9.(j), an instance `x1` of `X` accesses multiple instances of `Y` and multiple instances of `X` access multiple instances of Y. For these cases the code as shown in Fig. 4.13 must be generated for class `X`. The vector `destObjects` as shown in Fig. 4.14 is used to hold object references to multiple instances of `Y`.

```
class X
{
  Y destination;
}
```

FIGURE 4.12: Code generated for class `X` when the direction of data access is from left to right or bidirectional and when the cardinality ratio is `1:1` or `M:1`.

```
class X
{
  Vector destObjects;
}
```

FIGURE 4.13: Code generated for class `X` when the direction of data access is from left to right or bidirectional and when the cardinality ratio is `1:M` or `M:M`.

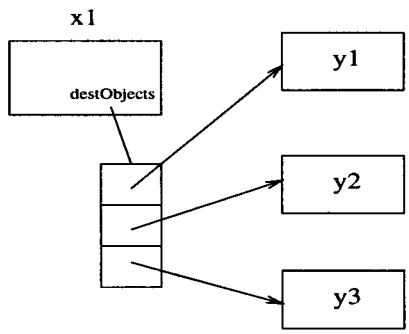

FIGURE 4.14: Vector `destObjects` of `x1` points to multiple instances of `Y`.

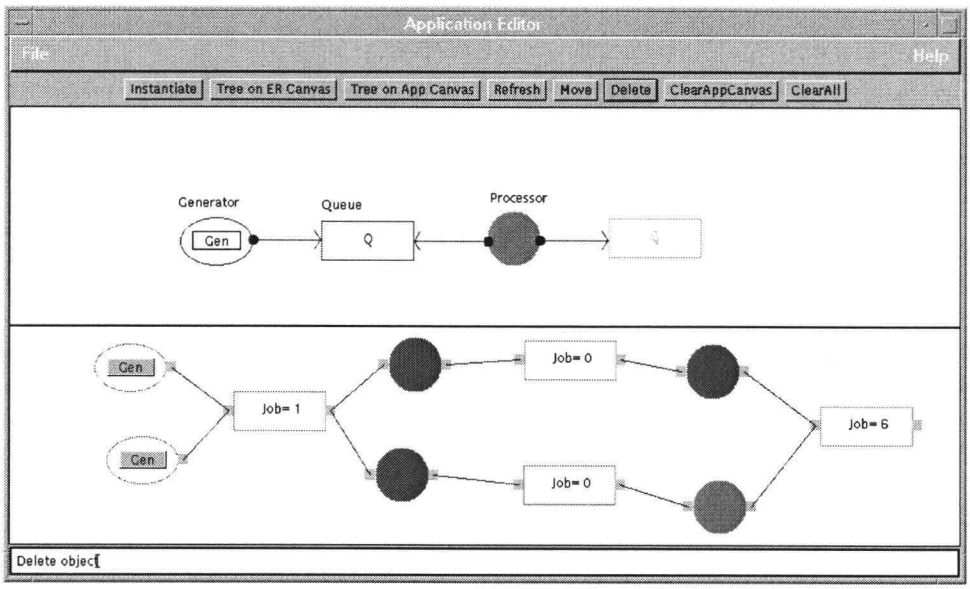

FIGURE 4.15: The application editor screen with a queueing system simulator application.

4.3 Application Editor

In this section we describe how to create an application by instantiating an entity from an EERD and by interconnecting them together. Fig. 4.15 shows the *application editor* screen with a queueing system simulator application. The application editor consists of

1. the upper canvas where an EERD is displayed,

2. the lower canvas where an application is composed,

3. a menu of buttons for editing operations in the third row from the top, and

4. the file and help menus in the second row from the top.

4.3.1 Creating Entities

We now describe how to create primitive and array entities with the application editor.

4.3.1.1 Primitive Entities

To create an instance of a primitive entity type, a user must take the following steps.

1. Enter the *entity-instantiation* mode by clicking a left mouse button with the cursor on the command button `Instantiate`.

2. Select the entity type on the EERD whose instance will be created by clicking the left mouse button with the cursor on it.

3. Click the left mouse button with the cursor at any location inside the lower canvas. When the left mouse button is released, the newly created entity is placed at the location where the mouse click occurred.

Now each time the left mouse button is clicked inside the lower canvas area, an instance of the selected entity type is created and placed at the location where the mouse click has occurred. Entities are active as soon as they are created.

4.3.1.2 Array Entities

To create an instance of an array entity type, a user must take the following steps.

1. Enter the *entity-instantiation* mode by clicking the left mouse button with the cursor on the command button `Instantiate`.

2. Select the array entity type on the EERD whose instance will be created by clicking the left mouse button on it.

3. Drag the cursor on the lower canvas while pressing the left mouse button for the distance of the size of the array entity to be created. When the left mouse button is released, the newly created array entity is placed on the canvas.

Now the user can create instances of the array entity type selected by repeating step 3 as many times as needed.

4.3.1.3 Record Entities

In the current implementation, record entities are not supported, although supporting them is not difficult.

4.3.2 Connecting Entities

Once entities are created, the user can connect these entities with each other. `PortViews` associated with an entity are created when the entity is instantiated. A `PortView`, which is a small rectangle attached to an entity, is a location where a user make a connection.

We now describe how to connect an entity to another entity with a relationship-by-arrow or relationship-by-proximity.

4.3.2.1 Relationships as Links

To connect two entities together with a link representing a relationship, take the following steps.

1. Enter the *entity-instantiation* mode by clicking the left mouse button with the cursor on the command button `Instantiate`.

2. Click the left mouse button with the cursor first inside a `PortView` of one entity and then inside a compatible `PortView` of another entity.

When the connection is successful, a relationship represented by a link is created between the two entities.

4.3.2.2 Relationships as Glue Strips

An entity can be connected to another one by placing them closely either under the *entity-instantiation* or under the *entity-move* mode. If an entity is instantiated and placed close to another compatible entity, a connection is established immediately. If an entity is instantiated, but if it is not placed close to another compatible entity, a connection is not established at this point. In this case, a user can establish a connection between two entities with the *entity-move* mode, which can be entered with the command button `Move`. In this mode, move one entity close to another compatible entity until their invisible `PortViews` overlap. An invisible `PortView` is a white `PortView` attached to the edge of each entity.

A success connection between two entities with a relationship by proximity results in a creation of a light gray glue strip between the two entities.

4.3.3 Interface Application

Interface `Application` shown in Fig. 4.16 includes the specification for the virtual function `createObject()`. This interface is generated by the schema editor and is used by the application editor. Every top-level application class, e.g., class `QueueSystem` shown in Fig. 4.6, must implement this interface. The method `createObject()` is used to create objects of any application domain in an application-independent manner. The argument `index` indicates the current selected entity type in the EERD displayed as the menu by the application editor.

```
interface Application
{
  public EditApp createObject(int index, EditObject object,
                              int x, int y);
}
```

FIGURE 4.16: Interface `Application`.

4.4 Examples

In the following subsections, we demonstrate the effectiveness of the ERSDE in three application domains: queueing systems, tank systems, and local-area-network systems.

4.4.1 Queuing Systems

We first describe a description of queueing systems and its applications.

4.4.1.1 EERD for Queueing Systems

The upper canvas in Fig. 4.17 shows the EERD for queueing systems. There are three entity types: `Generator`, `Queue`, and `Processor`. A proxy of entity type `Queue` is also used. Each `Generator` produces jobs and stores them in a `Queue`. A `Queue` holds jobs. Each `Processor` takes a job from a `Queue`, processes it, and passes the processed job to another `Queue`.

4.4.1.2 A Queueing System Simulator Application

The lower canvas in Fig. 4.17 shows a queueing system simulator application. The application consists of three `Generators`, five `Queues`, and five `Processors`. The first `Generator` is connected to a `Queue`. That `Queue` is connected to two `Processors`.

The other two `Generators` are connected to another `Queue`. That `Queue` is connected to another `Processor` and so on.

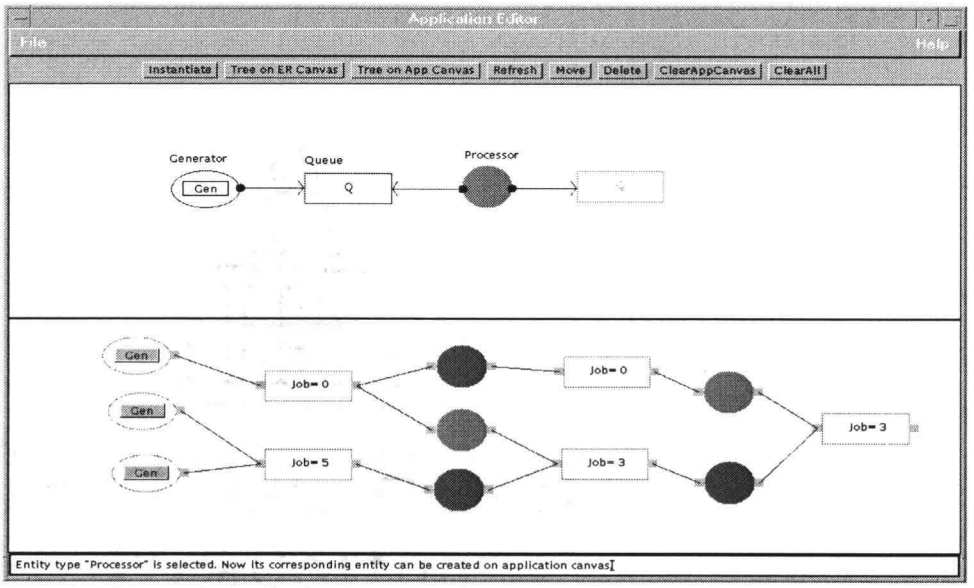

FIGURE 4.17: The application editor screen with a queueing system simulator application.

4.4.2 Tank Systems

We now describe tank systems applications.

4.4.2.1 EERD for Tank Systems

The upper canvas in Fig. 4.18 shows the EERD for tank systems. A tank system consists of tanks, valves, and pumps. A tank contains liquid, a pump makes liquid flow, and a valve controls the amount of flow. The liquid flows from left to right. The output end of a tank can be connected to the input ends of possibly multiple

valves, and the output end of a valve can be connected to the input end of either a pump or a tank. The output end of a pump can be connected to the input end of at most one valve, and the input end of a tank can be connected to the output ends of possibly multiple valves.

4.4.2.2 A Tank System Simulator Application

The lower canvas in Fig. 4.18 shows a tank system simulator application. The application is composed from two `Tanks`, four `Valves`, and two `Pumps`. The output of the first `Tank` is connected to the input `PortViews` of the two `Valves` in the left half of the canvas area, and the input of the second `Tank` is connected to the output `PortViews` of the two `Valves` in the right half of the canvas area. Each `Pump` is connected to the `Valves` on its left and right.

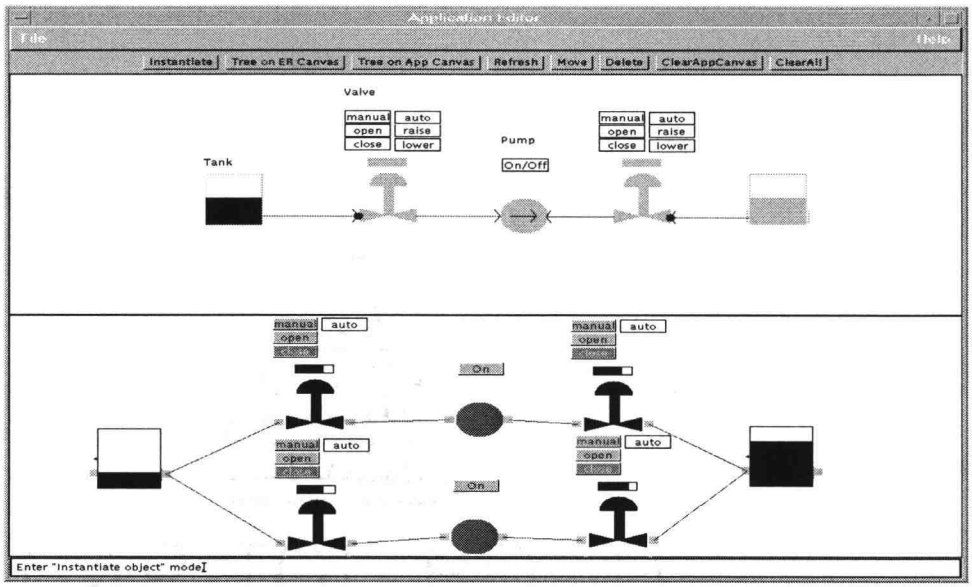

FIGURE 4.18: The application editor screen with a tank system simulator application.

4.4.3 Local-Area-Network Systems

We finally describe local-area-network system applications.

4.4.3.1 EERD for Local-Area-Network Systems

The upper canvas in Fig. 4.19 shows the EERD for local-area-network systems. A local-area-network system consists of `Cables`, `Stations`, and `Repeaters`. A `Cable` consists of multiple `Cable Segments`. A `Station` generates signals and send them to the `Cable Segment` to which it is connected. Each `Cable Segment` propagates signals to both left and right neighbors. A `Repeater`, then, propagates signals from one `Cable Segment` of one `Cable` to another `Cable Segment` of another `Cable`.

4.4.3.2 A Local-Area-Network System Simulator Application

The lower canvas in Fig. 4.19 shows a local-area-network system simulator application. The application consists of two `Stations`, three `Cables`, and two `Repeaters`. The first `Cable` consists of nine `Cable Segments`, the second one eleven, and the last one six. The leftmost `Station` is connected to the second `Cable Segment` of the leftmost `Cable`. The leftmost `Repeater` is connected to the seventh `Cable Segment` of the leftmost `Cable` and to the first `Cable Segment` of the middle `Cable`, and so on.

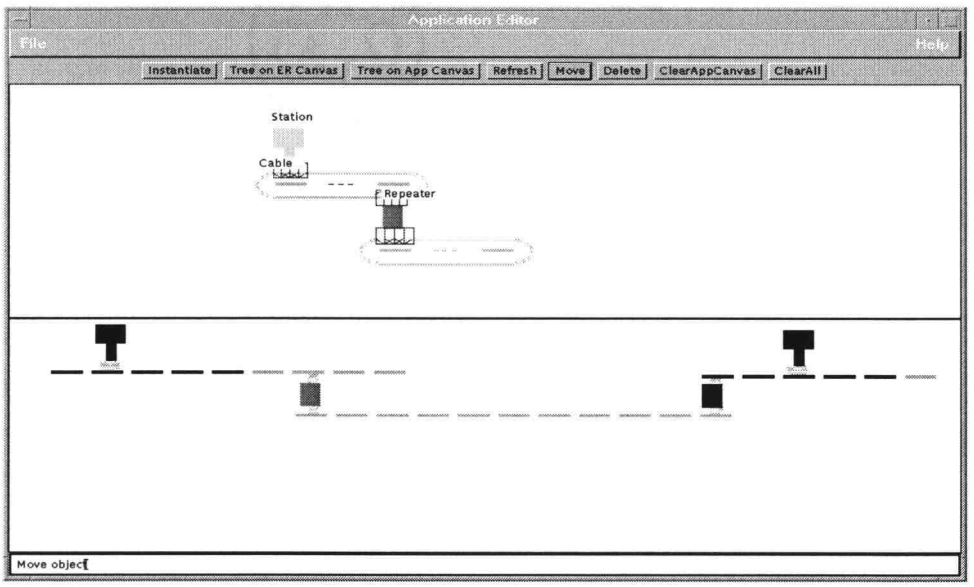

FIGURE 4.19: The application editor screen with a local-area-network system simulator application.

Chapter 5

IMPLEMENTATION DETAILS OF THE ERSDE

In this chapter, I describe the implementation details of the entity-relationship software development environment (ERSDE), including the *entity-type editor*, the *schema editor*, and the *application editor*. I implemented these three editors in Java and with Java Abstract Windowing Toolkit (AWT) on an HP-UX machine. Since Java programs are compiled into architecture-independent byte-code, the three editors can be executed on any machine that supports Java.

5.1 Implementation of the Entity-Type Editor

We use the entity-type editor to create entity types to be used by the schema editor. The graphical interface of the prototype of the entity-type editor is shown in Fig. 4.1. The component hierarchy of the entity-type editor is shown in Fig. 5.1. The major component of the entity-type editor is an `EditorFrame`. Class `EditorFrame` is a subclass of java AWT `Frame`.

`EntityTypesPanel`, `ColorPanel`, `ManipulatePanel`, `MenuBarPanel`, and `DrawingCanvas` are subclasses of java AWT `Panel`. The `EntityTypesPanel` contains a set of buttons for creating graphical objects representing entity types. The `ColorPanel` contains another set of buttons to be used to specify color for a graphical object. The `ManipulatePanel` contains a set of buttons for various commands for graphical editing. The `ClassMembers` dialog contains a set of `Choices`, and `Dialogs` and `TextFields` for collecting information class members of the entity type being created.

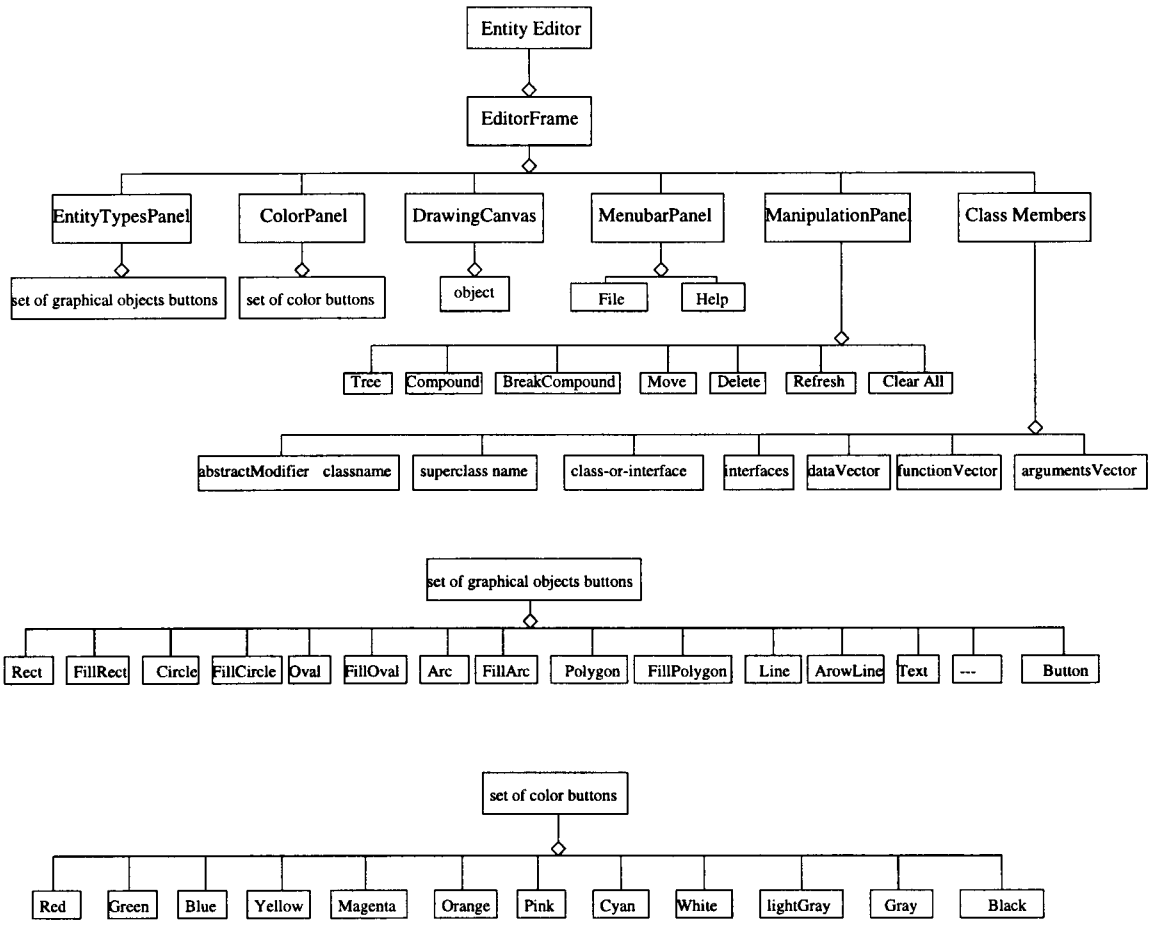

FIGURE 5.1: The component hierarchy of the entity editor.

The DrawingCanvas contains a Vector named objects whose members are the graphical objects being drawn on canvas. The DrawingCanvas supports methods paint(), mouseDown(), mouseMove(), mouseDrag(), and mouseUp(). These methods respond to mouse events and window events that occur on the canvas. The MenuBarPanel contains the File menu and the Help menu.

The entity-type editor supports such editing modes as Move, Delete, Compound, and BreakCompound in order to manipulate graphical objects displayed on the DrawingCanvas. The Move mode is used to relocate a selected graphical object, the Delete mode to remove a selected graphical object, the Compound mode to group multiple graphical objects into a compound graphical object, and the BreakCompound mode to decompose a compound graphical object back to its component graphical objects.

5.1.1 Creation of Graphical Objects

Every graphical object created by the entity-type editor is an EditObject. Fig. 5.2 shows the subclasses of class EditObject. Fig. 5.3 shows the data structure and member functions of class EditObject.

Each subclass of EditObject must override member functions paint(), is_on(), and draft(). The paint(Graphics g) method draws a graphical object on the canvas with a specified color, width and height.

A mouse event such as a mouse button down, up, or drag is detected by the DrawingCanvas. When a mouse down event occurs, the DrawingCanvas searches for a graphical object in which the mouse down event has occurred by invoking the method is_on(x, y) for each graphical object. Then, an appropriate editing operation is performed for the selected graphical object. For example, if the current

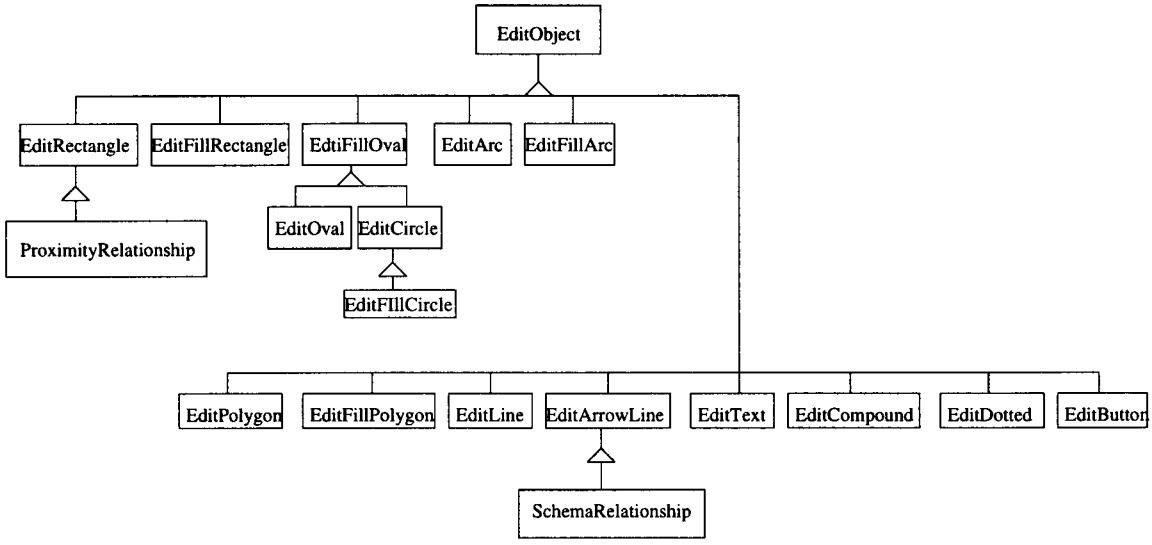

FIGURE 5.2: The `EditObject` class hierarchy.

```
class EditObject extends Canvas
{
  protected  int  x, y, w, h;     // xy-coordinates, width, height
  protected  Color  color;        // color
  public     DrawingCanvas canvas;
  Vector     information;         // information for an entity type

  public void  paint(Graphics g) {}
  public boolean  is_on(int x, int y) {return false;}
  public void  draft(Graphics g, int x1, int y1, int x2, int y2) {}
  public void  readData(DataInput in) throws IOException {}
  public void  writeData(DataOutput out) throws IOException {}
}
```

FIGURE 5.3: Class `EditObject`.

mode is `Delete`, the selected graphical object is deleted from the canvas when a succeeding mouse up event occurs.

The method `draft()` performs rubberbanding a graphical object while it is being drawn. If a subclass of `EditObject` supports additional member variables, some member functions accessing these variables, such as `read_data()` and `write_data()`, must be overridden.

5.1.2 Implementation of Class Members

The top row of the entity-type editor is where class information is collected. The modifier `Abstract` can be specified with `Checkbox`. The `class` and `superclass` names can be provided in the `TextFields`. Member variables and functions can be specified by clicking the button labeled `Class Members`. Then the `Class Members` dialog as shown in Fig. 4.1 is activated.

The `Class Members` dialog has three buttons labeled `Implements`, `Data Members`, and `Member Functions`. The `Implements` button activates the `Interfaces Information` dialog that is used to collect the names of the interfaces to be supported. The `Data Members` button activates the `Data Members Information` dialog that collects the access modifier, static modifier, final modifier, type, name, and initial value of each member variable.

The `Member Functions` button activates the `Member Functions Information` dialog, which is used to collect the access modifier, static modifier, final modifier, type, and name of each member function. The information takes effect when the `OK` button is clicked.

Each dialog allows ten entries to be filled by the user, but more entries can be added when the `More` button is clicked. The `More` button, which activates another dialog, allows ten more items to be added.

5.1.3 Implementation of File and Help Menu

The main menubar contains two menus: the `File` and `Help` menus. The `File menu` is a conventional file menu supporting `Save` and `Load` operations for files. An entity type can be stored in a file through a `DataOutputStream` and read from a file through a `DataInputStream`. We use the filename extension `.ety` for a file containing an entity type. An entity type created can be retrieved by the entity-type editor for modification. It can be read also by the schema editor while constructing an EERD.

The entity-type editor saves an entity type to a file as follows.

1. A `DataOutputStream` for an output file is created.

2. The size of `Vector objects`, which contains the graphical objects in the entity type, is recorded.

3. Each `EditObject` in `Vector objects` is asked to write the name of the class and the values of its data members with its method `writeData()`.

The entity-type editor reads an entity type from a file as follows.

1. A `DataInputStream` for an input file is created.

2. The number of graphical objects in the current entity type is read.

3. An instance of the class whose name is read from the input file is created with method `(Class.forName(className)).newInstance()`.

4. The instance of `EditObject` is added to `Vector objects`.

5. The `EditObject` is asked to read the values of its data members with its method `readData()`.

6. Steps 3-5 are repeated for every graphical object.

7. Each `EditObject` in `Vector objects` is asked to display itself on the canvas.

The `Help` menu provides information on how to use the entity-type editor. Method `setHelpMenu()` is used to put the `Help` menu at the rightmost position of the menubar.

5.2 Implementation of the Schema Editor

We use the schema editor to create EERDs to be used by the application editor. The graphical interface of the prototype of the schema editor is shown in Fig. 4.2. The component hierarchy of the schema editor is shown in Fig. 5.4. The main component of the schema editor is a `SchemaEditorFrame`. Class `SchemaEditorFrame` is a subclass of java AWT `Frame`.

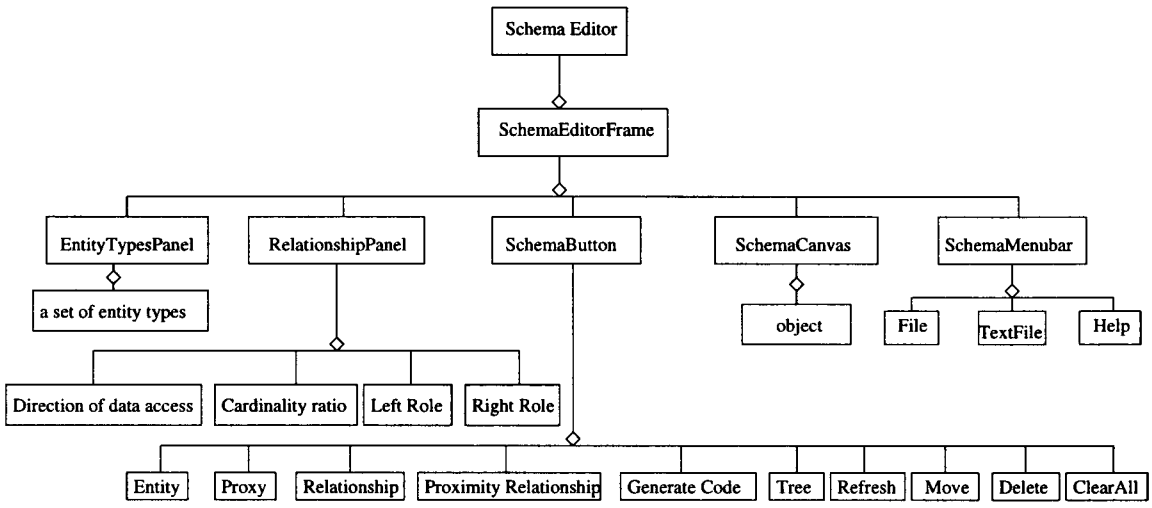

FIGURE 5.4: The component hierarchy of the schema editor.

`EntityTypesPanel`, `RelationshipPanel`, `SchemaButton`, `SchemaCanvas`, `SchemaMenubar` are subclasses of java AWT `Panel`. Class `EntityTypesPanel` is a subclass of java AWT `Panel`. The `EntityTypesPanel` contains an `entityList`, which is an instance of java AWT `List`. The `entityList` contains a list of available entity types. The names of the available entity types are loaded from a default file and added to the `entityList` when the schema editor is activated. A new entity type can be added to the `entityList` by using the `open()` command of `File` menu of the schema editor.

The `RelationshipPanel` contains `Choices` of the directions of data access and cardinality ratios, and `TextFields` for left and right roles of relationship types. The `SchemaButton` creates a set of buttons for graphical editing operations. The `SchemaCanvas`, where an EERD is constructed, is an instance of a subclass of `DrawingCanvas` discussed in section 5.1. The `SchemaMenubar` contains three menues: the `File` menu, the `TextFile` menu, and the `Help` menu.

5.2.1 Placement of Entity Types and Proxy Entity Types

A user can enter the *entity-type-placement* mode by selecting the command button `Entity`. In this mode, she can select the entity type to be placed on the `SchemaCanvas` from the `entityList`. When the left mouse button is clicked at a location inside the `SchemaCanvas`, the selected entity type is created by using the data read from a file through a `DataInputStream`.

A proxy-entity type is placed on the canvas if the original related entity type has already been placed on the canvas. For a proxy-entity type, only the fact that a graphical object is a proxy and its location need be recorded.

5.2.2 Creation of Relationship Types

We support two representations of relationship types: arrows and glue strips. These two forms of representation are semantically identical as stated in chapter 3. Fig. 5.2 shows a class hierarchy of these relationship types: `SchemaRelationship` and `ProximityRelationship`. A relationship type by an arrow is an instance of `SchemaRelationship`. `SchemaRelationship` is a subclass of `EditArrowLine`, which in turn is a subclass of `EditObject`. `SchemaRelationship` extends `EditArrowLine` by incorporating a direction of data access, a cardinality ratio, and roles.

A relationship type by proximity is an instance of `ProximityRelationship`. `ProximityRelationship` is a subclass of `EditRectangle`, which in turn is a subclass of `EditObject`. `ProximityRelationship` extends `EditRectangle` by incorporating a direction of data access, a cardinality ratio, and roles. Three small `EditArrowLines` in a `ProxyRelationship` indicate the direction of data access. We collect information on the direction of data access and the cardinality ratio of relationship type by using java `Checkboxes`.

The following three kinds of information used for code generation is associated with each relationship type: the cardinality ratio, the direction of data access, and the roles of the entities involved.

5.2.3 Implementation of File Menu

Class `EditObject` supports methods `readData()` and `writeData()` to store and read the data for an EERD. We use the filename extension `.er` for a file containing EERD data.

The schema editor saves an EERD to a file as follows.

1. A `DataOutputStream` for an output file is created.

2. The size of `Vector objects`, which contains the entity types and the relationship types in EERD, is written.

3. Each `EditObject` in `Vector objects` is instructed to write the name of the class and the values of its data members with its method `writeData()`.

The schema editor reads an EERD from a file as follows.

1. A `DataInputStream` for an input file is created.

2. The number of `EditObject` in the current EERD is read.

3. An instance of the class whose name is read from the input file is created with method `(Class.forName(className)).newInstance()`.

4. The instance of `EditObject` is added to `Vector objects`.

5. The `EditObject` is asked to read the values of its data members with its method `readData()`.

6. Step 3-5 are repeated for every `EditObject`.

7. Each `EditObject` in `Vector objects` is asked to display itself on the canvas.

5.2.4 Code Generation

When the EERD is completed, the schema editor produces two kinds of code: a top-level application-specific class and two classes, a model class and a view class, for each entity type displayed in the EERD. The schema editor also generates the code for the pointer structures for the relationship types, taking account of their cardinality ratios. The schema editor creates files to store code generated for those two kinds of code.

5.2.4.1 Top-Level Application-Specific Class

The schema editor generates the method `createObject()` in the top-level application-specific class. This method can create an instance of any entity type in the current EERD. Class `QueueSystem` shown in Fig. 4.6 is the top-level class for a queueing system.

5.2.4.2 Class for Model

The schema editor generates the following code for a model class.

1. **Data members and member functions**

   A model class contains its data members and the skeleton code for its member functions. The data members and the member functions must have been specified with the entity-type editor.

2. **Pointer structures**

   As we described in section 4.2.5.3, the schema editor adds pointer structures to the model class implementing an entity type. A pointer structure is created for each relationship type participated by the entity type.

5.2.4.3 Class for View

The schema editor generates the following code for a view class.

1. **A pointer to its model**

   An object reference to an instance of the model class is added to the view class.

2. **PortView**

   One `PortView` is added to an entity type for each connection with a relationship type. We discuss `PortView` in section 5.2.5.

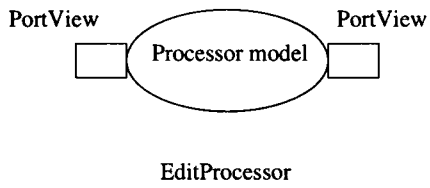

FIGURE 5.5: `EditProcessor`: a view of a `Processor`.

A view class is a subclass of `EditApp`. The name of a view class begins with `Edit` and is followed by the name of the entity type. Fig. 5.5 shows a graphical representation of an `EditProcessor` with two `PortViews`.

5.2.5 PortView

Three essential data members of `PortView` are the fields `multiplicity`, `role`, and `access`. Fig. 5.6 gives the major part of class `PortView`, whose complete code is given in Appendix C.

The code that instantiates a `PortView` is added to a view class by the schema editor. The schema editor uses the direction of data access, the cardinality ratio, and the role of a relationship type to set the initial values of the fields `multiplicity`, `access`, and `role` of the `PortViews` associated with that relationship type.

5.2.5.1 Multiplicity

If the relationship type is `one-to-one`, the `multiplicity` field of a `PortView` associated with each end of the relationship type is set to `SINGLE`. If the relationship type is `one-to-many` or `many-to-one`, the `multiplicity` field of a `PortView` associated with the `many` side of the relationship type is set to `SINGLE` and that associated with the `one` side is set to `MULTIPLE`. If the relationship type is `many-to-many`, the

multiplicity fields of the PortViews associated with the relationship type are both set to MULTIPLE.

5.2.5.2 Role

We check the compatibility of two PortViews by using the type of the entities to which the PortViews are attached and the roles of the PortViews. The role of each PortView is obtained from either the left or right role of a relationship type. The left role of a relationship type is specified as text in field Left Role when the relationship type is created with the schema editor, and the right role of a relationship type is specified as text in field Right Role.

5.2.5.3 The direction of data access

The direction of data access may be OUT or IN. The value OUT indicates that the entity to which this PortView is associated has object references to the entities related. The IN indicates that the entity to which this PortView is associated is accessed by the entities related.

5.2.6 Compatibility of PortViews

The application editor connects two entities by using two compatible PortViews. Two PortViews are compatible if there is a relationship type in the EERD such that the two entity types connected by it have the types and roles specified by the selected PortViews. That is, PortView X with $< typeX, roleX >$ and PortView Y with $< typeY, roleY >$ are compatible if $< typeX, roleX >$ is the type and the role at one end of some relationship type and if $< typeY, roleY >$ is the type and the role at the other end of that relationship type.

Legal connections of type-role pairs in the EERD shown in Fig. 5.7.(a) are $(< A, role_1 > < B, role_2 >)$ and $(< A, role_3 > < C, role_4 >)$. Fig. 5.7.(b) shows

```
class PortView extends EditFillRectangle implements Constants
{
  int           multiplicity;      // SINGLE or MULTIPLE
  int           role;              // specified with the schema editor
  int           access;            // OUT or IN
  AObject       address;           // one active object
                                   // when multiplicity is SINGLE
  Vector        objects;           // a vector if MULTIPLE
  EditApp       editApp;           // point back to its entity type
  int           numberOfPortConnected = 0;
  Vector        linkVector;        // number of links of this PortView

  public PortView(int role, AppCanvas canvas,
                  EditApp editApp, int multiplicity) {}
  public Link connectPorts(PortView portView)
  {
    Link link = new Link();        // connects two PortViews
    boolean success = link.connectPorts(this,
                                portView, arrow, canvas);
  }
  public void link(PortView portView)
  {
    if (multiplicity == SINGLE)           // AObject of this PortView
       this.address.link(this, portView); // points to AObject
    if (multiplicity == MULTIPLE)         // add AObject of portView
       objects.addElements(portView.address);  // to Vector objects
  }
}
```

FIGURE 5.6: Class PortView.

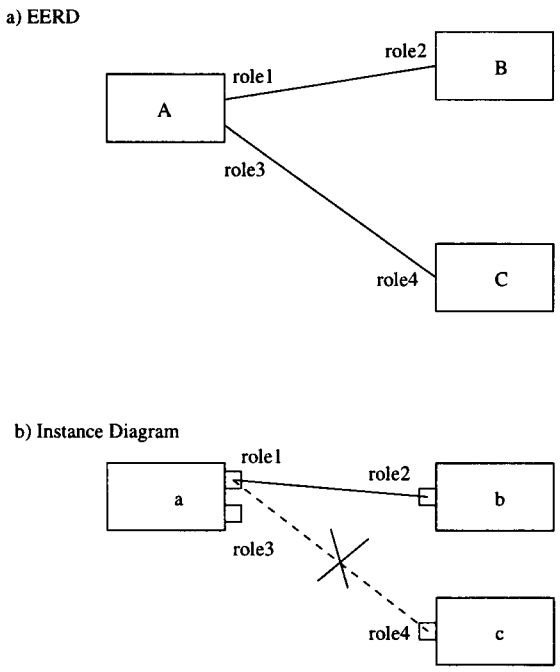

FIGURE 5.7: Legal connections.

entity a with $role_1$ can be connected to entity b with $role_2$, but it cannot be connected to entity c with $role_3$ because connection $(< A, role_1 > < B, role_2 >)$ is compatible, but connection $(< A, role_1 > < C, role_4 >)$ is not.

The application editor detects errors if the following illegal connections are attempted.

1. The user selects two incompatible `PortViews`.

2. Two compatible `PortViews` are selected, but the constraint on the cardinality ratio will be violated. That is, the permissible number of connection for the entity associated with one of the `PortViews` is one, and it has already been connected to some entity.

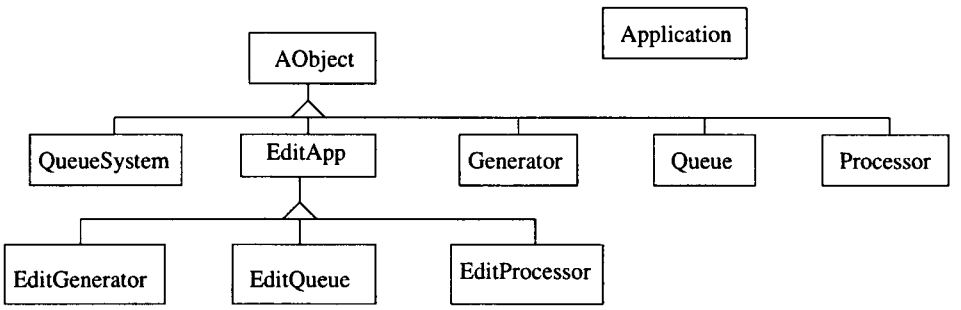

FIGURE 5.8: Classes generated from the EERD for queueing systems.

When a user clicks the left mouse button with the cursor on the second compatible `PortView`, the method `connectPorts()` of `PortView` is invoked, a `Link` that connects the two `PortViews` is created. We discuss class `Link` in section 5.3.

5.2.7 Example

For queueing system applications, seven files are generated by the schema editor: `QueueSystem`, `EditGenerator`, `Generator`, `EditQueue`, `Queue`, `EditProcessor`, and `Processor`. `QueueSystem` is the top-level application-specific class. `Generator`, `Queue`, and `Processor` are the classes for the model and `EditGenerator`, `EditQueue`, and `EditProcessor` are those for the view. The code generated for this example is discussed in section 4.2.5.1 and is shown in Appendix A.

Fig. 5.8 shows the class hierarchies of the classes used by queueing system applications. Class `EditApp` shown in Fig. 5.10 is a subclass of `AObject`. Each instance of `AObject`, which is a class in the Structural Active-Object System (`SAOS`), is an active object. `EditApp` is a superclass of every view class. For example, `EditGenerator` shown in Fig. 5.8 is a subclass of `EditApp`. `EditGenerator` is the view class of `Generator`. Fig. 5.9 shows the component hierarchy of the `EditGenerator`.

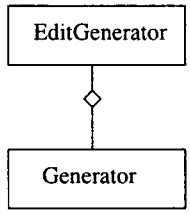

FIGURE 5.9: The `EditGenerator` component hierarchy.

```
class EditApp extends AObject implements Constants {
  EditCompound view;           // a view of this object
  AppCanvas    canvas;         // a canvas contains this object
  String   name;               // a name of this object
  int      objectID = -1;      // this object identification
  int      x, y;               // xy-coordinates of this object's view
  boolean  compositeObject = false;// true for a composite object
                               // false - not a composite object
  Vector   components;         // use to store components of
                               // a composite EditApp
  boolean  primitive = true;   // every entity is primitive by default
}
```

FIGURE 5.10: Class `EditApp`.

5.3 Implementation of Application Editor

With one application editor, we can compose applications in different domains by switching EERDs used as templates. The graphical interface of the prototype of the application editor is shown in Fig. 4.15. The component hierarchy of the application editor is shown in Fig. 5.11. The major component of the application editor is an `AppEditorFrame`. Class `AppEditorFrame` is a subclass of java AWT `Frame`.

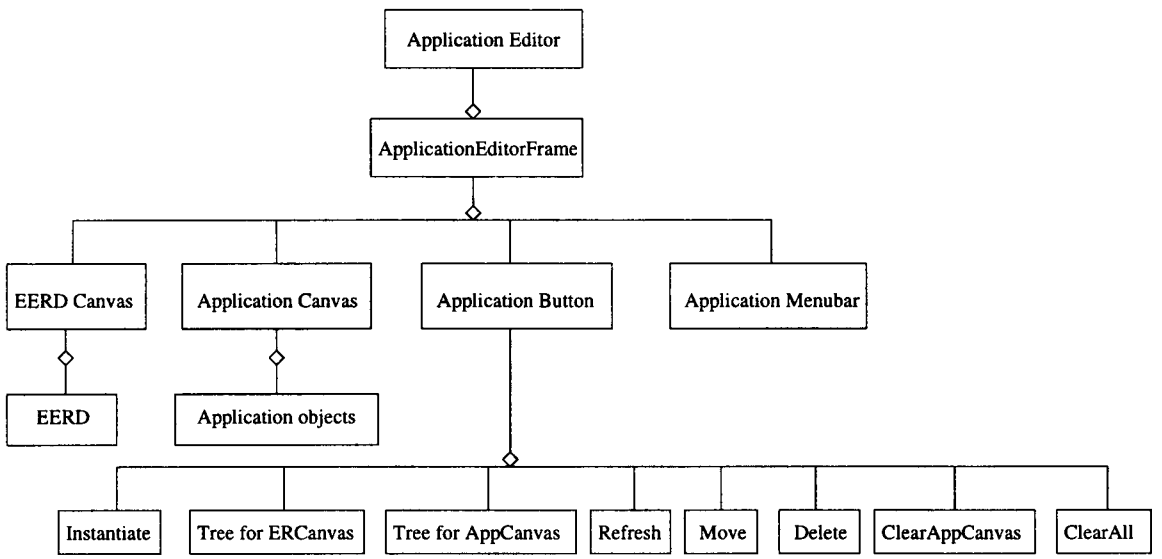

FIGURE 5.11: The component hierarchy of the application editor.

`ERCanvas`, `AppCanvas`, `AppButton`, and `AppMenubar` are subclasses of java AWT `Panel`. The `ERCanvas` is the upper canvas where an EERD is displayed. The `ERCanvas` detects the entity type selected. The `AppCanvas` is the lower canvas where an application is composed. The `AppCanvas` methods provides `paint()`, `mouseDown()`, `mouseDrag()`, and `mouseUp()`. Method `mouseUp()` supports three

modes: *entity-instantiation*, *entity-move*, and *entity-deletion* modes. The *entity-instantiation* mode allows entities to be created and placed on the lower canvas. The *entity-move* mode allows an entity to be moved to another location. If the entity moved is involved in a relationship by proximity it may be connected to other entities by glue strips. The `AppButton` contains a set of buttons used to activate editing operations. The `AppMenubar` contains two menues: the `File` menu and `Help` menu.

We now explain how two entities are connected. The dashed-line in Fig. 5.12 indicates a connection from `Generator` G1 to `Queue` Q1. The reference to `Queue` Q1 is set to the field `output` of `Generator` G1. To establish such a connection in the model, a `Link` that connects two `PortViews` are used by the application editor. A `Link` contains the references to the two `PortViews` that it connects.

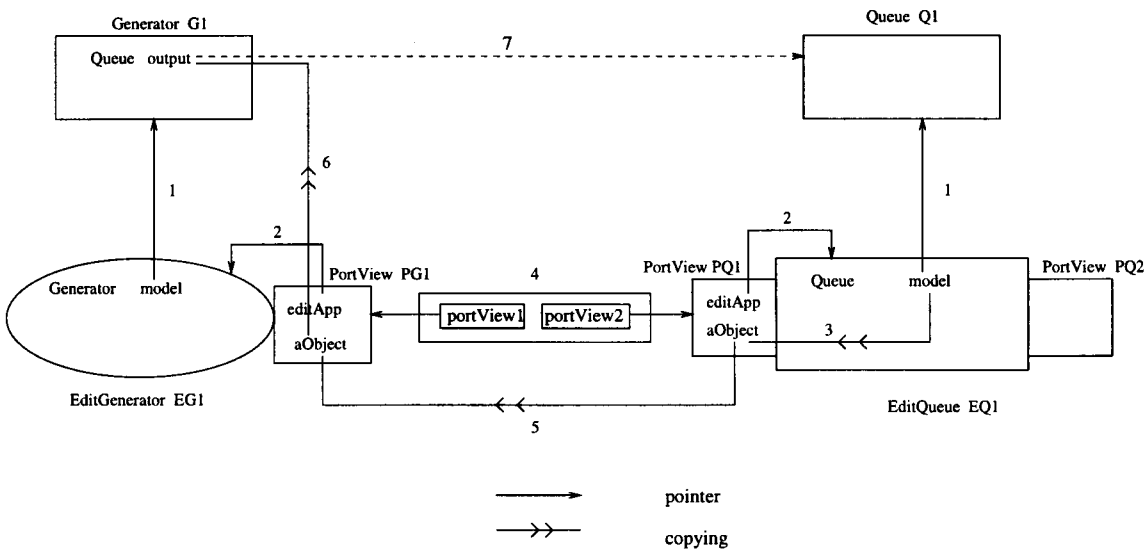

FIGURE 5.12: A `Link` connecting a `Generator` and a `Queue`.

In Fig. 5.12, the numbers associated with the arrows indicate the order of the steps taken to connect two entities in the model.

Step 1: Connecting the Model to the View

After `EditGenerator` EG1, `Generator` G1, `EditQueue` EQ1, and `Queue` Q1 are created, the object reference to `Generator` G1 is set to field `model` of `EditGenerator` EG1, and the one to `Queue` Q1 is set to field `model` of `EditQueue` EQ1.

Step 2: Connecting the Portview to the View

In step 2, the object reference to `EditGenerator` EG1 is set to field `editApp` of `PortView` PG1, and the one to `EditQueue` EQ1 is set to field `editApp` of `PortView` PQ1.

Step 3: Copying from `model` of EQ1 to `aObject` of PQ1

In step 3, the object reference to `Queue model` is copied to field `aObject` of `PortView` PQ1, and hence field `aObject` of `PortView` PQ1 will contain a reference to `Queue` Q1.

Step 4: Creating the Link

After the user clicks the left mouse button with the cursor first on `PortView` PG1 of `EditGenerator` EG1 and then on `PortView` PQ1 of `EditQueue` EQ1, a `Link` is created. The object reference to `PortView` PG1 is set to field `portView1` of the `Link`, and the one to `PortView` PQ1 is set to field `portView2` of the `Link`. The `Link` then makes the following connections.

Step 5: Copying PG1 to PQ1

In step 5, the object reference in field `aObject` of `PortView` PQ1 is copied to field `aObject` of `PortView` PG1. Consequently, field `aObject` of `PortView` PG1 will contain a reference to `Queue Q1`.

Step 6: Copying `output` of `Generator G1` to `aObject` of PG1

In step 6, the object reference in field `aObject` of `PortView` PG1 is copied to filed `output` of `Generator G1`. Thus, `Generator G1` will contain a reference to `Queue Q1`.

Step 7: Connecting field `output` of `EditGenerator` EG1 to `Queue` Q1

As a consequence of the preceding steps, field `output` of `Generator G1` points to `Queue Q1`.

A complete code for class `Link` is given in Appendix C.

Chapter 6

EVALUATION

We demonstrated in the previous chapter feasibility of the entity-relationship software development environment (ERSDE) by showing prototype applications created with it. This chapter reports the results of experiments performed to evaluate the effectiveness of the ERSDE in terms of comprehensibility of design documents and ease of composition of applications.

6.1 Experiment I: Comprehensibility of Design Documents

The purpose of this experiment was to determine which type of diagram, an Object Modeling Technique (OMT) class diagram or an extended entity-relationship diagram (EERD), helps programmers better understand an application domain and compose correct applications. Another question to be resolved was whether programmers preferred to use EERDs or OMT class diagrams as templates for constructing applications.

We formulated two hypotheses to be tested in this experiment. As our first hypothesis, we expected more subjects using EERDs as templates would compose correct applications than subjects using OMT class diagrams as templates. In order to test this hypothesis, we asked subjects in one group to develop applications in two different domains by using OMT class diagrams as templates and the subjects in another group to develop applications in the same application domains by using

EERDs as templates. For each group we recorded the total number of the subjects who composed applications correctly.

As our second hypothesis, we expected that the subjects would prefer to use EERDs as templates for composing applications than to use OMT class diagrams. In order to test this hypothesis, we showed the subjects OMT class diagrams and EERDs for two application domains and asked the subjects which diagram in each application domain they preferred.

6.1.1 Subjects

The subjects in this experiment were 50 undergraduate computer science students at Oregon State University. Seventeen of them were taking a junior-level introduction to software engineering (CS361) class. The other 33 students were taking a senior-level database management (CS440). The experiments were conducted one week before the final week of the Spring term, 1997. The CS361 subjects had studied the Booch notations. The CS440 students had studied the entity relationship (ER) modeling technique. As these two groups of students had different backgrounds, we analyzed their data separately.

6.1.2 Design of the Experiment

The subjects in each class were divided into two groups. The CS361 students were ranked according to their midterm scores. The student with the highest score was assigned to Group 1. The student with the second highest score was assigned to Group 2. The student with the third highest score was assigned to Group 2. The student with the fourth highest score was assigned to Group 1, and so on. Since the number of students in CS440 was large, we grouped them randomly into Group 1 and Group 2.

We briefly explained notations and provided a handout (See Appendix D) used in OMT class diagrams to the students in Group 1 and explained notations and provided a handout (See Appendix D) used in EERDs to the students in Group 2. The students were then asked to compose applications in two application domains, namely, tank systems and local-area network systems. The students in Group 1 in each class composed applications by using OMT class diagrams, and the students in Group 2 in each class composed both applications by using EERDs. Each subject had ten minutes to compose each application.

After the subjects finished composing both applications, they were shown both the OMT class diagrams and the EERDs, in each application domain. The subjects were asked which diagram in that domain they preferred to use as a template for composing an application.

6.1.3 Materials

All subjects worked with hardcopy of the materials. Appendix D contains all the materials used in this experiment, including

1. An explanation of the notations. Group 1 received the OMT class diagrams and Group 2 received an explanation of the EERD notations.

2. Group 1 received description of a sample queueing system application composed according to an OMT class diagram and Group 2 the EERD version,

3. A problem statement to construct a tank system application,

4. A problem statement to construct a local-area network system application, and

5. A question that asked each student her preference between OMT diagrams and EERDs for each application.

6.1.4 Procedure

The experiment was conducted as follows:

1. The subjects in Group 1 received a brief explanation of the notations used by OMT class diagrams, and the subjects in Group 2 those used by EERDs. They then read the handout.

2. Group 1 subjects saw the OMT class diagram for queueing systems and a queueing system application based on it. The EERD for the same application domain and the same queueing system application was shown to the subjects in Group 2.

3. The subjects in Group 1 were given the OMT class diagram for tank systems and were asked to compose one application in this application domain. The subjects in Group 2 were given the EERD for the same application domain and were asked to compose one application in that application domain. Subjects were told that the application should consist of at least five tanks, eight valves, and four pumps.

4. The subjects in Group 1 were given the OMT class diagram for local-area network systems and were asked to compose one application in this application domain. The subjects in Group 2 were given the EERD for the same application domain and were asked to compose one application in that application domain. Subjects were told that the application should consist of at least three cables, two repeaters, and two stations. Also each cable should have at least five cable segments.

5. After the previous two tasks were completed, each subject was shown both the OMT class diagrams and the EERDs for tank systems and was asked which she preferred and why. Next subjects were shown the OMT class diagrams and EERD for local-area network systems and were asked for their preference.

6.1.5 Analyses

6.1.5.1 Data Collected to Test the First Hypothesis

An application was graded correct (1) or incorrect (0). The application was correct if it was composed of the required number of components and the connections among components conformed to the permissible relationships shown in the OMT class diagram or the EERD. Otherwise, it was incorrect. We then tabulated the numbers of the students who composed applications correctly and incorrectly using OMT class diagrams or using EERDs for each class CS361 and CS440.

6.1.5.2 Data Collected to Test the Second Hypothesis

For the preference question, we counted the number of the students in each class who preferred OMT class diagrams and the number of those who preferred EERDs.

6.1.6 Results

6.1.6.1 Application Correctness

Table 6.1 summarizes the results for the tank system application. Eight of the students used the OMT class diagram for tank systems as a template, and nine students the EERD. As can be seen from Table 6.1 the two groups in CS361 class performed nearly the same and hence there was no significant difference. The same was true for the two groups in CS440 class. Most of the CS361 subjects who failed to construct the applications correctly did not follow the constraints on cardinality

ratios and made wrong connections for the tank system application. Most of the CS440 subjects who did not construct the application correctly made wrong connections.

One possible reason for similar performance on the tank problem is the closeness to the queueing system application. The OMT class diagram and the EERD for tank systems are not complex, and the OMT class diagram can be understood with basic training.

	OMT		EERD	
CS361	*Correct*	*Incorrect*	*Correct*	*Incorrect*
# subjects	3	5	3	6
%	37	63	33	67
CS440				
# subjects	4	13	7	9
%	24	76	44	56

TABLE 6.1: Result of the compositions of tank systems.

Table 6.2 summarizes the results for the local-area network system application. For this application significantly more subjects in both the CS361 and CS440 classes composed a correct solution using EERD (For CS361, Fisher's exact test $p = 0.012$; CS440, Fisher's exact test $p = 0.004$). Most of the CS361 and CS440 subjects who used an OMT class diagram as a template failed to make correct connections among entities.

	OMT		EERD	
CS361	*Correct*	*Incorrect*	*Correct*	*Incorrect*
# subjects	1	7	7	2
%	12	88	78	22
CS440				
# subjects	4	13	13	3
%	24	76	81	19

TABLE 6.2: Result of composition of a local-area network system.

6.1.6.2 Diagram Preference

Recall that for the diagram preference the subjects were shown OMT and EERD diagrams for each application and asked "Which one of these two diagrams do you think is better?" after they had worked both problems. The two diagrams shown to the subjects, the OMT class diagrams for tank systems and for local-area network systems and the EERDs for the same application domains, represent the same information; i.e., they are semantically identical. Their preferences are shown in Table. 6.3. Note that not all students completed the preference questions.

A significant number of students in each class preferred EERD for each application. (Tank system, CS361, One-sample sign test, $p = 0.0066$; Tank system, CS440, One-sample sign test, $p = 0.0129$; LAN system, CS361, One-sample sign test, $p = 0.0461$; LAN system, CS440, One-sample sign test, $p = 0.0069$).

Table 6.4 and Table 6.5 gives the reasons why they preferred EERD or OMT class diagram. The CS361 and CS440 students who preferred the EERDs thought

	CS361					
Domains	Tank Systems			LAN Systems		
Methods	OMT	EERD	No Response	OMT	EERD	No Response
# of subjects	2	12	3	3	10	4
%	14	86	-	23	77	-
	CS440					
Domains	Tank Systems			LAN Systems		
	OMT	EERD	No Response	OMT	EERD	No Response
# of subjects	8	21	4	7	21	5
%	28	72	-	25	75	-

TABLE 6.3: Preference for the diagraming methods.

that EERDs were clearer, easier to understand, more informative about connections, and more intuitive. A small number of the subjects who preferred OMT class diagrams said that OMT diagrams were less complicated and provided with labels more information on components and connections.

Table 6.6 shows the preferences of the subjects for diagraming methods in two application domains. The first column is the preferred diagram for tank system and the second column is that for LAN system. From Table 6.6, eight CS361 and seventeen CS440 students preferred EERDs for both applications. More than half of the subjects in each class preferred to use EERDs in composing applications in both domains.

# of subjects	Domains					
	Tank Systems			LAN Systems		
Methods	EERD	OMT	No Response	EERD	OMT	No Response
Easy to understand	5	-		3	-	
Use of iconic	1	-		1	-	
Realistic	1	-		-	-	
Clearer	-	-		2	-	
More information on connections	4	-		2	1	
Less confusing	-	1		-	-	
Intuitive	1	-		2	-	
Better syntax	-	-		-	1	
No reason	-	1		-	1	
Total	12	2	3	10	3	4

TABLE 6.4: Reasons for the preference of diagraming methods for CS361 students.

# of subjects	Domains					
	Tank Systems			LAN Systems		
Methods	EERD	OMT	No Response	EERD	OMT	No Response
Easy to understand	8	6		15	4	
Use of iconic	3	-		1	-	
Realistic	-	-		-	-	
Clearer	2	-		1	-	
More information on connections	3	1		2	2	
Less confusing	2	-		-	1	
Intuitive	2	-		1	-	
Better syntax	-	-		-	-	
No reason	1	-		1	-	
Simpler	-	1		-	-	
Total	21	8	4	21	7	5

TABLE 6.5: Reasons for the preference of diagraming methods for CS440 students.

Diagram Preferences		Number of Subjects			
Tank Systems	LAN Systems	CS361	%	CS440	%
EERD	EERD	8	53	17	57
EERD	OMT	2	13	4	13
OMT	EERD	1	7	3	10
OMT	OMT	1	7	3	10
EERD	-	2	13	0	0
-	EERD	1	7	1	3
OMT	-	0	0	2	7
-	OMT	0	0	0	0
-	-	2	-	3	-
Total		17		33	

TABLE 6.6: Preferences for diagraming methods in two application domains. The dash (-) indicates "no response".

6.2 Experiment II: Comparison of the ERSDE and Menu-Based Editors

In this experiment, we wanted to determine whether the ERSDE application editor is more effective in composing applications than menu-based structural active-object system (SAOS) editors.

We formulated four hypotheses to compare the effectiveness of such methods.

1. We expected more subjects who would compose correct applications by using the ERSDE application editor than by using the menu-based SAOS graphical editors. In order to test this hypothesis, subjects in one group developed applications in two different domains with the menu-based SAOS graphical editors and the subjects in another group developed applications in the same application domains with the ERSDE application editor.

2. We anticipated that the subjects would compose applications faster with the ERSDE application editor than with the menu-based SAOS graphical editors. We measured this by the time required by a subject to compose each application.

3. We expected that the subjects would perform more tasks in composing applications with the menu-based SAOS graphical editor than with the ERSDE application editor. To test this hypothesis, we observed the number of mouse clicks made by a subject in composing each application. The number of mouse clicks for each subject was recorded in two stages: when the subjects created a `Cable` and when the subjects made a connection using glue strip. If the subjects could not create a `Cable` or connect entities using glue strip, the number of mouse clicks was also recorded until they gave up.

4. We expected that more subjects would prefer the ERSDE application editor than the menu-based SAOS graphical editors in composing applications. In order to test this hypothesis, we demonstrated the ERSDE application editor and the menu-based SAOS graphical editors to the subjects after they had worked both problems and asked them which editor they preferred for each application domain.

6.2.1 Subjects

The subjects in this study were 18 computer science graduate students at Oregon State University. The backgrounds of the subjects are summarized in Table 6.7. The subjects were randomly divided into two groups. The subjects in one group worked with menu-based SAOS graphical editors and those in another group with the ERSDE application editor.

Environment	Years in CS		OO experience		Component experience	
	mean	median	yes	no	yes	no
SAOS	4	3	5	4	6	3
ERSDE	5.4	5	5	4	7	2
Total			10	8	13	5
Percentage			56	44	72	28

TABLE 6.7: Summary of the backgrounds of the subjects for Experiment II.

6.2.2 Design of the Experiment

Subjects were divided into two groups according to their experience in object-oriented approaches. Each group had about the same number of the subjects who had experience in object-oriented approaches. After training, the subjects composed applications in two application domains: queueing systems and local-area network systems. The subjects in Group 1 were asked to compose applications with the menu-based SAOS graphical editor and the subjects in Group 2 were asked to compose both applications with the ERSDE application editor.

6.2.3 Procedure

Experiment II was conducted as follows:

1. Training

   Each subject in each group was instructed how to compose a tank system application with the given editor. Each subject in Group 1 was given a hands-on introduction to the menu-based SAOS graphical editor and each subject in Group 2 a hands-on introduction to the ERSDE application editor. Each subject was taught until he understood how to use the designated graphical editor. The EERD for tank systems were displayed on the upper canvas of the ERSDE application editor and the available entity types are displayed on the upper canvas of menu-based SAOS graphical editors.

2. Working on the First Problem

   Each subject in Group 1 was given the menu-based SAOS graphical editor for queueing systems and a set of requirements for a queueing system application they were to develop. Each subject in Group 2 was given the ERSDE

application editor with the EERD for the same application domain and a set of requirements for a queueing system application they were to develop. Each subject had as much time as needed to compose the application. The time used by each subject and the difficulties experienced by each subject were recorded.

3. Working on the Second Problem

   Each subject composed a local-area network system application by repeating step 2.

4. Editor Preference

   After finishing composing both applications, each subject was shown how to compose the same applications with the other graphical editor. That is, the subjects in Group 1 were shown the ERSDE application editor, and those in Group 2 the menu-based SAOS graphical editors. We then asked in which editor they preferred composing both applications, menu-based SAOS graphical editors or the ERSDE application editor.

6.2.4 Analyses and Data Collected

An application were graded as correct (1) or incorrect (0). The application was correct if it was composed of the required number of components and it was executable. Otherwise, it was incorrect. We recorded the time required by each subject to compose an application. We also recorded the number of mouse clicks made by the subjects to create entities and to connect entities together while they were composing an application. Finally we recorded each subject editor preference.

6.2.5 Results

6.2.5.1 Application Correctness and Time Required to Compose an Application

Table. 6.8 shows the number of correct and incorrect queueing system and local-area network system applications, and Table. 6.9 summarizes the times needed to create them. From these results, we can make the following observations.

1. All the subjects could compose queueing system applications correctly. The tools used, the menu-based SAOS graphical editor or the ERSDE application editor, did not affect the results significantly. This may be due to the fact that composing a queueing system application was similar to composing the tank system application which was used in the training example.

2. For a queueing system application, all the subjects who used the ERSDE application editor attempted to incorporate different configurations of connections, while only 67% of the subjects who used the menu-based SAOS graphical editor did so. Of those who tried to incorporate different configurations of connections, 33% of the subjects in each group successfully incorporated these connections.

3. For a local-area network application, 67% of the subjects composed applications correctly with the ERSDE application editor, while only 33% of the subjects did so with the menu-based SAOS graphical editor. However, this difference is statistically still not significant (Fisher's exact test, $p = 0.172$).

4. The subjects took significantly longer to compose queueing system applications with the menu-based SAOS graphical editor than with the ERSDE graphical editor (Queue program: Mann-Whitney test, $p = 0.01$; LAN program: Mann-Whitney test, $p = 0.046$).

Queuing System	SAOS		ERSDE	
	Correct	*Incorrect*	*Correct*	*Incorrect*
# of Subjects	9	0	9	0
%	100	0	100	0
Local-Area Network System				
# of Subjects	3	6	6	3
%	33	67	67	33

TABLE 6.8: Correctness of the queueing system applications composed.

One possible reason that the subjects who used the menu-based SAOS graphical editor took longer composing both the queueing system applications and the local area network system applications may be because of the extra time needed to read the textual information from the online help menu. This help information described how to create components and how to make connections between them. The subjects who used the ERSDE graphical editor could see the patterns of the relationships among components from the EERD, while the subjects who used the SAOS graphical editor did not get this information from the screen and had to access the help information.

The subjects in both groups who failed to construct the LAN applications correctly either did not create `Cables` correctly or did not establish correct connections with glue strips between components.

We also investigated whether the backgrounds of the subjects affected their performance in composing applications or not. Table 6.10 shows for the LAN problem the subjects who composed applications correctly and incorrectly grouped according

Queueing System	Environments	
Time	SAOS	ERSDE
Average (std)	8.78 (2.44)	6.00 (2.29)
Min	5	2
Max	11	9
LAN System		
Average (std)	10.44 (3.21)	7.67 (2.96)
Min	5	4
Max	15	14

TABLE 6.9: Times (in minutes) used to compose queueing system applications.

to their object-oriented, component composition, and their computer science experience. None of the differences was significant.

6.2.5.2 Graphical Editor Preference

Table. 6.11 summarizes the graphical editor preferences of the subjects. It turned out that every student preferred the ERSDE application editor to the menu-based SAOS graphical editors. All subjects preferred the ERSDE because of the following reasons.

1. The ERSDE had the EERD of an application domain displayed graphically in the upper canvas. This EERD helped the students understand the relationships among components easily, clearly, and intuitively.

OO Knowledge	*Correct*	*Incorrect*
Yes	4	6
No	5	3
Composition	*Correct*	*Incorrect*
Yes	6	7
No	3	2
Years in CS	*Correct*	*Incorrect*
1-3	3	5
4+	6	4

TABLE 6.10: The performance based on the backgrounds of the subjects

2. As expected, the EERD allowed subjects to visualize an application domain quickly. In other words, the subjects did not want to read textual descriptions provided by the online help facility.

6.2.5.3 Difficulties Experienced

We were also interested in the difficulties encountered by the subjects while they composed applications, particularly when they created and connected components. Table. 6.12 shows the number of incorrect attempts made by the subjects while they tried to create `Cables` and to connect other components to them for LAN problem. Table. 6.13 shows the number of subjects according to their difficulties experienced in composing each application.

	Menu-Based SAOS Graphical Editor	ERSDE Graphical Editor
# of of Subjects	0	18
%	0	100

TABLE 6.11: Preference for the environments.

1. The subjects who used either graphical editor did not have trouble creating components of queueing system applications. We believe this was because creating components for the queueing system applications similar to the way the components of the example tank system applications were created.

2. The subjects experienced a minor problem while connecting components for queueing system applications. The permissible cardinality ratio between

   `Generators` and `Queues` is *many-to-one*, meaning that multiple `Generators` can be connected to one `Queue` but that a `Generator` can not be connected to multiple `Queues`. However, some subjects tried to connect a `Generator` to multiple `Queues`. A similar problem also occurred when the subjects tried to make connections between `Queues` and `Processors`. These mistakes were quickly detected and corrected since both editors gave error messages.

   From Table. 6.13, although more subjects using menu-based SAOS graphical editor did not try to incorporate cardinality ratios into their applications than the subjects using the ERSDE application editor, the difference was not statistically significant (Fisher's exact test, $p = 0.10$). Of those who tried to

SAOS			ERSDE		
Subject #	Creation	Connection	Subject #	Creation	Connection
1	1	4	10	1	4
2	2	6	11	2	5
3	5	10+	12	3	3
4	8	(20)	13	5	6
5	10	(5)	14	6	4
6	10	(7)	15	9	3
7	(5)	(8)	16	(6)	(0)
8	(7)	(10)	17	(7)	(12)
9	(20+)	(19)	18	(10+)	(10+)

TABLE 6.12: Numbers of incorrect attempts. A number without () means the number of incorrect attempts before a subject successfully created a local-area network systems application. A number in () means the number of incorrect attempts before a subject stopped without a correct final result.

incorporate cardinality ratios, the same proportion of the subjects (33 %) in each group were successfully.

3. The subjects had a major difficulty in creating components for local-area network system applications. For example, a `Cable` was a composite entity which consists of an array of a variable number of `Cable Segments`, and the subjects had to drag the left mouse button in the lower canvas as far as the length of the `Cable`. Many subjects did not understand this manipulation.

Difficulties Experienced		
while incorporating cardinality ratios for Queueing Systems	SAOS	ERSDE
Did not try	3	0
Tried, but failed	4	6
Tried and success	2	3
while creating `Cables` and using glue strips for LAN Systems		
Could not create a cable	3	3
Created cables, but could not make connections	3	0
Created cables and made correct connections	3	6

TABLE 6.13: Difficulties experienced while the subjects composed applications.

From Table. 6.13, the same proportion of the subjects (33 %) in each group could not create a `Cable`.

4. The subjects had another major problem in connecting components together by gray glue strips. Connecting two components by a gray glue strip could be accomplished only when the subjects placed one component sufficiently close to the other one so that their `PortViews` overlapped. The subjects did not realize that they had to know the positions of the `PortView` of each component in order to make a correct connection. They simply moved an entity close to the other entity without making their `PortViews` overlap.

Of those subjects who successfully created the `Cables`, half of the subjects using menu-based SAOS graphical editor could not make connection with glue strips while all the subjects using the ERSDE application editor made the connection, the difference was not statistically significant (Fisher's exact test, $p = 0.09$).

6.3 Summary of Results

We obtained the following conclusions from Experiments I and II.

1. A significant number of the subjects who used the EERD composed local-area network system applications correctly, while only a small number of the students who used the OMT class diagram did so. However, little difference was observed for tank system applications.

2. Most of the subjects preferred using EERDs to OMT class diagrams as design documents.

3. Although more students composed applications correctly with the ERSDE application editor than with the menu-based SAOS graphical editors, this difference was not statistically significant.

4. The subjects took significantly longer time to compose applications with the menu-based SAOS editors than with the ERSDE editor.

5. All the 18 subjects preferred using the ERSDE application editor to the menu-based SAOS graphical editors as a software development environment.

6. The subjects experienced more difficulties with menu-based SAOS graphical editors than with the ERSDE application editor.

Chapter 7

CONCLUSIONS AND FUTURE RESEARCH

In concluding this thesis, I summarize the results of my research, and suggest some future research topics.

7.1 Conclusions

We presented a software development environment for composing application software from components. This software development environment, called the *Entity-Relationship Software Development Environment* (ERSDE), uses extended entity-relationship diagrams (EERDs) as templates for application software. Its three major facilities are the *entity-type editor*, the *schema editor*, and the *application editor*.

The entity-type editor allows us to create entity types to be used in EERDs. We can use the schema editor to create a domain-specific schema as an EERD by placing entity types and then interconnecting them with relationship types. Once the EERD for an application domain is complete, the schema editor can generate skeleton code for the classes for the application domain represented by the EERD. The classes representing the entity types include the pointer structures implementing the relationship types.

We can compose an application with the application editor by instantiating entities from the entity types in the EERD for that application and then connecting

those entities according to the relationship patterns shown in the EERD. Then, if behavior definitions are manually attached to the entity types with the schema editor, the application created from the EERD becomes executable.

We introduced some new ideas and notations for EERDs: *iconic entity types*, *proxy entity types*, and *relationship representation by proximity*. One new idea used by the ERSDE is to use an EERD as the main 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, iconic representations of entity types in the EERD enhances its intuitiveness.

We implemented prototypes of the entity-type editor, the schema editor, and the application editor. With the entity-type editor and the schema editor, we created the EERDs for queueing systems, tank systems, and local-area network systems. With the application editor and with these EERDs, we successfully created applications for a queueing system, a tank system, and a local-area-network system.

In order to evaluate the effectiveness of the ERSDE, we conducted two experiments and obtained the following results.

1. The proportion of the subjects who composed local area network system applications correctly with the EERDs for this application domain was larger than the proportion of the students who did so with the OMT class diagrams in the same application domain.

2. Using an EERD and using an OMT class diagram did not make any difference in creating tank system applications.

3. The subjects preferred EERDs to OMT class diagrams as templates for constructing applications.

4. The graphical editor used, the menu-based SAOS graphical editor or the ERSDE graphical editor, did not affect the correctness of the programs constructed.

5. The subjects took significantly longer to compose applications with the menu-based SAOS editor than with the ERSDE editor.

6. All the subjects preferred the ERSDE editor to the SAOS editor in composing queueing system and local-area-network system applications.

We demonstrated that we can construct an executable application program from its components by connecting them as indicated by the extended entity-relationship diagram in that application domain.

7.2 Future Research

Some research subjects left for future research are as follows.

1. The current system supports only centralized applications. The scope of the ERSDE should be extended to include distributed applications. One way to do this extension is to use *distributed structural object composition* (DSOC) [48].

2. In the current system, legal connections among entities are restricted by relationship types. Each relationship type can be considered to define a pattern involving two entity types. To allow more complicated rules for configurations, we need to support patterns each of which contains more than two entity types and one relationship type.

3. It is desirable to generate more code for some well-known design patterns such as `Composite` objects, and `Observables/Observers` [31].

4. The entity subclassing is not implemented in the current system. We should support entity subclassing.

We now discuss some details of these future research topics.

7.2.1 Distributed Applications

DSOC is a software construction mechanism for distributed application development. In DSOC, the designer of a distributed application can obtain distributed objects that interact properly with each other simply by instantiating them from their classes and then by composing (interconnecting) them structurally. Interactions among cooperative distributed objects are abstracted and programmed into the classes for those objects. An EERD can be used to show possible configurations for DSOC applications.

Gundoju implemented the *distributed observable/observer* mechanism to construct Model/View-Controller (M/VC)-based distributed applications by DSOC [39]. This mechanism is designed to simplify implementation of distributed systems that require instantaneous presentations of remote data. When the state of a *distributed* (or *remote*) *observable* changes, it sends messages notifying this change to all the *distributed* (or *remote*) *observers* associated with it. Upon receiving these notification messages, the distributed observers update their graphical representations. An application may contain multiple *server* sites, which contain distributed observables, and multiple *client* sites, which contain distributed observers.

7.2.2 General Patterns

A pattern in general should be allowed to include any number of entity types and relationship types. Assume that we have the EERD shown in Fig. 4.18 for tank systems. A good engineering practice requires that whether two Tanks are connected

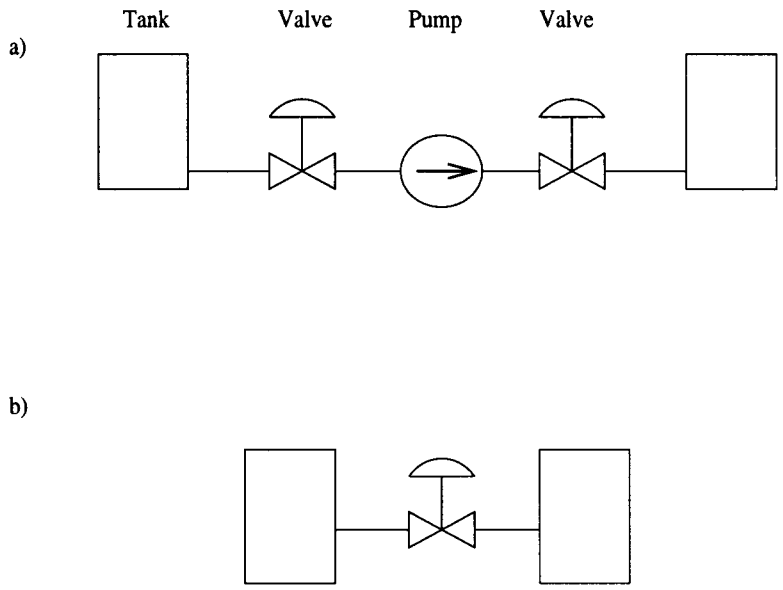

FIGURE 7.1: Connection of two Tanks.

there must be a Pump between them with one Valve at each end of the Pump and a Tank as shown in Fig. 7.1.a. However, according to the current EERD rules, the EERD allows undesirable configurations. That is, as shown in Fig. 7.1.b, two Tanks can be connected without a Pump between them.

7.2.3 Code Generation for Predefined Patterns

For a known pattern, we can generate more code than simple pointer structures.

7.2.3.1 Composite Objects

A composite object contains other objects, and it can be treated as one object. Fig. 7.2 shows the record entity type X with components of entity types A, B, and C. A class for composite objects should support the following capabilities: adding a component, removing a component, and returning an enumerator for the components. The method `addComponent(Object object)` adds `object` to the composite

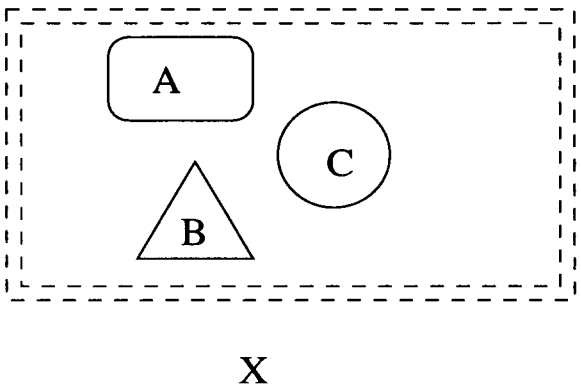

FIGURE 7.2: A record entity type X with components of entity types A, B, and C.

object and the method `removeComponet(Object object)` removes `object` from the composite object. The method `getEnumerator()` returns an `Enumerator` for the components of the composite object.

We consequently can generate for class `EditX` the code for methods `addComponent(Object object)`, `removeComponet(Object object)`, and `getEnumerator()` as shown in Fig. 7.3.

7.2.3.2 Observables/Observers

An Observable/Observer mechanism defines one-to-many dependency among objects so that when one object changes its state, all its dependents are notified. On receipt of this notification, each dependent object updates its state. Fig. 7.4 shows an observable x with observers a, b, c, and d. When the state of x changes, x notifies its observers a, b, c, and d. Each observer then updates its state.

An `Observable/Observer` relationship type is an extension of a one-to-many relationship type. Java supports class `Observable` and interface `Observer` according to the model-view paradigm. An instance of a subclass of `Observable` maintains a set

```
Class EditX {
  Vector components;       // holds its components
  A a;                     // component a
  B b;                     // component b
  C c;                     // component c
  public EditX()
  {
    a = new A();
    b = new B();
    c = new C();
    components.addElement(a);  // add a to components
    components.addElement(b);  // add b to components
    components.addElement(c);  // add c to components
  }
                               // add object to components
  void addComponent(Object object)
  {
    components.addElement(object);
  }
                               // remove object from components
  boolean removeComponet(Object object)
  {
    return components.removeElement(object);
  }
                               // get enumerator for the components
  VectorEnumerator getEnumerator()
  {
    return new VectorEnumerator(components);
  }
}
```

FIGURE 7.3: Class EditX

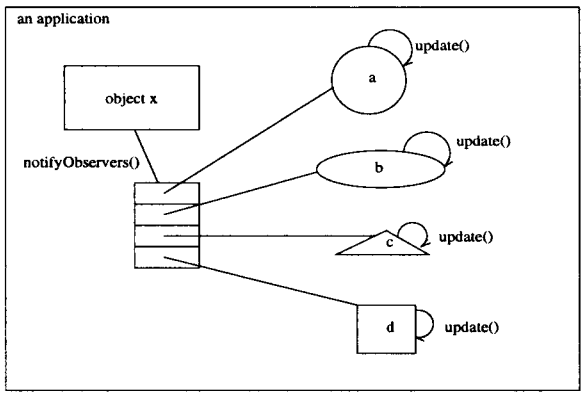

FIGURE 7.4: `Observable` x and `Observers` a, b, c, and d.

of `Observers`. Class `Observable` supports methods `addObserver(Object object)`, `deleteObserver(Object object)`, and `notifyObservers(Object arg)`, and interface `Observer` requires method `update()`.

The method `addObserver(Object object)` adds `Observer object` to the set of `Observers` associated with the `Observable`, and the `deleteObserver(Object object)` deletes `Observer object` from the set of these `Observers`. When the instance of a subclass of `Observable` changes its state, it invokes the method `notifyObservers(Object arg)`. The method `notifyObservers(Object arg)` notifies all the `Observers`. Each `Observer` then invokes the method `update(Observable object, Object arg)` provided for that object. The Java code as shown in Fig. 7.5 can be generated for classes Application, X, and A.

7.2.4 Entity Subclassing

The notation of entity subclassing is shown in Fig. 7.6. 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

```
class Application
{
  X x;                  // observable x
  A a;                  // observer a
  B b;                  // observer b
  C c;                  // observer c
  D d;                  // observer d
  public Application()
  {
    x = new X();
    a = new A();
    b = new B();
    c = new C();
    d = new D();
    x.addObserver(a);  // add observer a to observable x
    x.addObserver(b);  // add observer b to observable x
    x.addObserver(c);  // add observer c to observable x
    x.addObserver(d);  // add observer d to observable x
  }
}

class X extends Observable
{
                              // to be filled by user
}

class A implements Observer
{
  public A(){}
                              // automatically invoked when
                              // its observable changes
  public update(Observable object, Object arg)
  {
                              // to be filled by user
  }
}
```

FIGURE 7.5: Skeleton code for an Observable/Observer application.

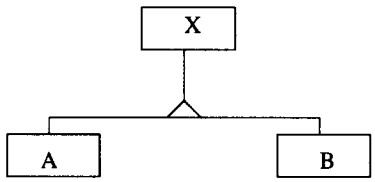

FIGURE 7.6: Entity subclassing.

entity types. With the entity subclassing we can generate more code for subtypes. The current ERSDE prototype does not support entity subclassing, although doing so is not difficult.

BIBLIOGRAPHY

[1] AVS Technical Overview, October 1992.

[2] ActiveX Overviews, July 1996.

[3] JavaBeans 1.0 API specification. Technical report, December 1996.

[4] *"Tutorial: Using the Beans Development Kit 1.0"*. Java Soft, February 1997.

[5] Gregory Abowd, Robert Allen, and David Garland. Using Style to Understand Description of Software Architecture. In *Proceeding of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 18, pages 9–20, Los Angeles, California, USA, December 1993. ACM SIGSOFT.

[6] Richard M. Adler. "Emerging Standards for Component Software". *IEEE Computer*, pages 68–76, March 1995.

[7] J.E. Archer and M.T. Devlin. "Rational's Experience Using Ada for Very Large Systems". In *First International Conf. Ada Programming Language Applications for the NASA Space Station, NASA*, pages B2.5.1–B2.5.12, June 1986.

[8] Kent Beck and Ward Cunningham. "A Laboratory for Teaching Object-Oriented Thinking". In *Conference on Object-oriented Programming Systems, Languages, and Applications; reprinted in Sigplan Notices*, volume 24, pages 1–6. ACM, October 1989.

[9] Ted Biggerstaff and Charles Richter. "Reusability Framework, Assessment and Directions". *IEEE Software*, pages 41–49, March 1987.

[10] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[11] Grady Booch. Object Oriented Development. *IEEE Trans. on Software Engineering*, 12(2), February 1986.

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

[13] K. Brockschmidt. *"Inside OLE 2.0"*. Microsoft Press, Richmond, Wash., 1994.

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

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

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

[17] 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.

[18] Curtis R. Cook, Jean C. Scholty, and Jame C. Spohrer, editors. *Empirical Studies of Programming: Fifth Workshop.* Wadsworth Publishing Company, 1993.

[19] Steve Cook and John Daniels. *"Designing Object Systems: Object-Oriented Modelling with Syntropy.* The Object-Oriented series. Prentice Hall International (UK) Ltd., 1994.

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

[21] P. Cox and T. Pietrzykowshi. "Using a Pictorial Representation to Combine Dataflow and Object-orientation in a Language Independent Programming Mechanism". In *International Computer Science Conference*, pages 695–704, 1988.

[22] Wayne W. Daniel. *Applied Nonparametric Statistics*, chapter 2-3, pages 27–31,82–87. Houghton Mifflin Company, 1978.

[23] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann. "Software Development Environments". *IEEE Software*, pages 18–28, November 1987.

[24] C.J. Date. *"A Note on One to One Relationship"*, chapter 16, pages 427–450. Relational Database Writings 1985-1989. Addison-Wesley, 1990.

[25] Vicky de May. *Visual Composition of Software Application*, pages 276–303. In Nierstrasz and Tsichritzis [63], 1995.

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

[27] K. Anders Ericsson and Herbert A. Simon. *"Protocol Analysis: Verbal Reports As Data"*. The MIT Press, revised edition, 1993.

[28] Robert G. Fichman and Chris F. Kemerer. "Object-oriented and Conventional Analysis and Design Methodologies: Comparison and Critique". *IEEE Computer*, 25(10):22–39, October 1992.

[29] Jr. Frederick P. Brooks. *"The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Publishing Company, anniversary edition, 1995.

[30] Alfonso Fuggetta. "A Classification of CASE Technology". *IEEE Computer*, pages 25–38, December 1993.

[31] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *"Design Patterns, Elements of Reusable Object-Oriented Software"*. Addison-Wiley, 1995.

[32] David Garland, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceeding of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 19, pages 175–188, New Orleans, Louisiana, USA, December 1994. ACM SIGSOFT.

[33] David Garland, Robert Allen, and John Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *Proceedings 17th International Conference on Software Enginneering*, pages 179–185, Seattle, Washington, April 1995. ACM SIGSOFT.

[34] David Garland and Dewakne E. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transaction on Software Engineering*, 21(4):269–274, April 1995.

[35] David Garland and Mary Shaw. *"An Introduction to Software Architecture"*, chapter 1, pages 1–39. World Scientific, 1993.

[36] A. Goldberg. *"The Influence of an Object-oriented language on the Programming Environment"*, pages 141–174. McGraw Hill, New York, 1984.

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

[38] D. Goodman. *"The Complete Hypercard Handbook"*. Bantam Books, 1988.

[39] Vikram Gundoju. Distributed observable/observer: A distributed object-communication mechanism. Technical report, Dept. of CS, Oregon State University, September 1997.

[40] Robert Helsel. *"Graphical Programming: A Tutorial for HP VEE.* Prentice Hall PTR, 1995.

[41] Watts S. Humphrey. *"A Discipline for Software Engineering"*, chapter 1, page 1. Addison-Wesley, 1995.

[42] 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.

[43] J. Johnson, B. Nardi, C. Zarmer, and J. Miller. "ACE: Building Interactive Graphical Applications". *Communications of the ACM*, 36(4):41–55, April 1993.

[44] Anker Helms Jorgensen. *"Using the Thinking-Aloud Method in System Development"*. In Salvendy and Smith [73], 1989.

[45] M. Kass. "CONDOR: Constraint-Based Dataflow". In *SIGGRAPG'92*, pages 321–330, 1992.

[46] Haim Kilov and William Harvey, editors. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1996.

[47] J. Kramer, J. McGee, M. Sloman, SC. Cheung N. Dulay, S. Crane, and K. Twidle. "An Introduction to Distributed Programming in REX". In *Esprit 1991 Conference*, pages 207–221, 1991.

[48] Chih Lai, Tonghyun Lee, Toshimi Minoura, and Chee-Hang Park. Distributed structural active-object system (dsaos) for groupware implementation. In *1995 Pacific Workshop on Distributed Multimedia Systems.* Prentice Hall, 1995.

[49] David Alex Lamb. *"Software Engineering: Planning for Change"*. Prentice Hall, 1988.

[50] Ronald J. Leach. *"Software Reuse: Methods, Models, and Costs"*, chapter 1-4, pages 1–138. McGraw-Hill, 1997.

[51] Adrienne Y. Lee and Nancy Pennington. *"Learning Computer Programming: A Route to General Reasoning Skills?"*, pages 113–121. In Cook et al. [18], 1993.

[52] T. G. Lewis. *"CASE: Computer-Aided Software Engineering"*. Van Nostrand Reinhold, 1991.

[53] Yanbing Lu. Structural Active-Object Systems in Java. Technical report, Dept. of Computer Science, Oregon State University, September 1996.

[54] C. McClure. *"CASE is Software Automation"*. Prentice Hall, 1989.

[55] Toshimi Minoura. *Structural Active-Object Systems Fundamentals*, chapter 9, pages 143–162. In Kilov and Harvey [46], 1996.

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

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

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

[59] David E. Monarch and Gretchen I. Puhr. "A Research Typology for Object-oriented Analysis and Design". *Communication of ACMs*, 35(9):35–47, September 1992.

[60] Pornsiri Muenchaisri and Toshimi Minoura. Software Composition with Extended Entity-Relationship Diagrams. In *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems*, pages 113–127. USENIX, June 1996.

[61] J. Neighbors. "The Draco Approach to Constructing Software from Reusable Components". *IEEE Trans. on Soft. Eng.*, SE-10(5):564–574, September 1984.

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

[63] Oscar Nierstrasz and Dennis Tsichritzis. *Object-Oriented Software Composition.* The object-oriented series. Prentice Hall, 1992.

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

[65] Xavier Pintado. "Gluons: a Support for Software Component Cooperation". Technical report, University of Geneva, 1991.

[66] Xavier Pintado. *"Gluon and the Cooperation between Software Components"*, pages 321–351. In Nierstrasz and Tsichritzis [63], 1995.

[67] John W. Pratt and Jean D. Gibbons. *Concepts of Nonparametric Theory*, chapter 5, pages 238–241. Springer-Verlag, 1981.

[68] Roger S. Pressman. *"Software Engineering: A Practitionaer's Appproach"*, chapter 1, pages 1–20. McGraw-Hill, 2 edition, 1997.

[69] Fred L. Ramsey and Daniel W.Schafer. *"The Statistical Sleuth: A Course in Methods of Data Analysis"*. An International Thomson Publishing Company, 1997.

[70] Trygve Reenskaug. *"Working with Objects: The OOram Software Engineering Method.* Manning Publication Co., Greenwich, 1996.

[71] Steven P. Riess. "PECAN: Program Development Systems that Support Multiple Views". *IEEE Trans. on Soft. Eng.*, SE-11:276–285, March 1985.

[72] 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.

[73] G. Salvendy and M.J. Smith, editors. Elsevier Science Publishers B.V., Amsterdam, 1989, the Netherlands, 1989.

[74] Stephen R. Schach. *"Classical and Object-Oriented Software Engineering"*. Irwin, 3 edition, 1996.

[75] 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.

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

[77] Mary Shaw. Architecture Issues in Software Reuse: It's Not Just Functionality, It's the Packaging. In *Symposium on Software Reusability*, page 3. ACM SIGSOFT, April 1995.

[78] 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.

[79] Mary Shaw and David Garlan. *Software Architecture: Perspective on an Emerging Discipline.* Prentice Hall, 1996.

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

[81] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software Architecture in Industrial Applications. In *Proceedings 17th International Conference on Software Enginneering*, pages 196–207, Seattle, Washington, USA, April 1995. ACM SIGSOFT.

[82] D.C. Swinehart, P.T. Zellweger, and R.B. Hagmann. "The Structure of Cedar". *SIGPlan Notices*, pages 230–244, July 1985.

[83] W. Teitelman and L. masinter. *"The Interlisp Programming Environment"*, pages 73–81. Computer Society Press, Los Alamitos, Calif., 1981.

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

[85] Daniel Tkach, Walter Fang, and Andrew So. *"Visual Modeling Techniques: Object Technology Using Visual Programming.* Addison-Wiley, 1996.

[86] Benjamin Volozh, Mari Kopp, and Enn Tyugu. NUT Graphics. January 1997.

[87] Iseult White. *"Using the Booch Method: A Rational Approach.* The Benjamin/Cummings Publishing Company, 1994.

[88] C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Greene. "ITS: A Tool for Rapidly Developing Interactive Applications". *ACM Trans. on Info. Sys.*, 8(3):204–236, September 1990.

[89] Rebecca Wirfs-Brock and Brian Wilkerson. "Object-oriented Design: A Responsibility-Driven Design". In *Conference on Object-oriented Programming Systems, Languages, and Applications; reprinted in Sigplan Notices*, volume 24, pages 71–76. ACM, October 1989.

[90] Chukiat Worasucheep. Software Channels: A Distributed Object-Communication Mechanism . Technical report, Dept. of Computer Science, Oregon State University, September 1996.

APPENDICES

Appendix A

SKELETON CODE GENERATED

This appendix shows the code for some generated classes such as `EditGenerator`, `Generator`, `EditQueue`, `Queue`, `EditProcessor`, and `Processor`.

```
class EditGenerator extends EditApp implements Constants
{
  Generator  model;
  Button11   button;
  public final int ENABLE = 0;
  public final int DISABLE = 1;

  public EditGenerator(EditCompound object, AppCanvas canvas,
         int x, int y)
  {
    super(x, y);
    model = new Generator("Generator");
    view = (EditCompound) object.clone();
    this.canvas = canvas;
    name = new String("Generator");
    outPortView = new PortView(OUT, canvas, this, SINGLE);
  }
  public void initialize() throws SaosException
  {
    insert(model, true);
    int diffX = x - view.x;
    int diffY = y - view.y;
    view.updateCoordinates(x,y, diffX, diffY);
    view.setColor(Color.green);
    view.setCanvas(canvas);
    view.initialize(canvas.getGraphics());

    outPortStatus = false;
    portNumber = 2;
    outPortView.initialize();
  }
  public void setPtr(PortView port) throws SaosException
  {
    if (port.role == OUT)
      model.setOutput(port);
  }
}
```

```
public class Generator  extends  AObject
{
  public static final int GIdle = 0;
  public static int GBusy = 1;
  public static final int GReady = 2;
  public static final int START = 0;
  public static final int GENERATE = 1;
  public static Random rand = new Random();
                                  // use the current time to seed
  public Queue  output;           // link to a queue
  public ABoolean enabled;
  public AInteger state;
  public int interval = 8;
  public int counter;
  public void initialize () throws SaosException {}
  public void setOutput(PortView port) throws SaosException {}
  public void start () throws SaosException {}
  public void generate () throws SaosException {}
  public void dispatch(int funcIndex) throws SaosException
  {
    switch (funcIndex)
    {
      case START:    break;
      case GENERATE: break;
    }
  }
}
```

```
class EditQueue extends EditApp implements Constants
{
  public final int UPDATE = 0;
  Queue  model;
  Label  label;

  public EditQueue(EditCompound object,
                   AppCanvas canvas, int x, int y)
  {
    super(x, y);
    model = new Queue("Queue");
    view  = (EditCompound) object.clone();
    this.canvas = canvas;
    name = new String("Queue");
    label = new Label("Job= 0");
    inPortView = new PortView(IN, canvas, this, SINGLE);
    outPortView = new PortView(OUT, canvas, this, SINGLE);
  }
  public void paint(Graphics g) {}
  public void initialize() {}
  public void dispatch(int funcIndex) {}
  public void ifupdated() {}
}
```

```
public class Queue extends AObject
{
  public int nSlots = 30000;
  public AInteger nJobs;
  public static final int REDRAW = 0;

  public Queue(String name)
  {
    super(name);
    nJobs = new AInteger(0);
  }
  public void print (int level) {}
  public void initialize () {}
  public void paint (Graphics g) {}
  public void redraw () {}
  public void dispatch (int funcIndex) {}
}
```

```
class  EditProcessor  extends  EditApp implements Constants
{
  public final int PIdle = 0;
  Processor  model;

  public EditProcessor(EditCompound object,
                       AppCanvas canvas, int x, int y)
  {
    super(x, y);
    model = new Processor("Processor");
    view  = (EditCompound) object.clone();
    this.canvas = canvas;
    name = new String("Processor");
    inPortView = new PortView(IN, canvas, this, SINGLE);
    outPortView = new PortView(OUT, canvas, this, SINGLE);
  }

  public void initialize() throws SaosException
  {
    insert(model, true);
    int diffX = x - view.x;
    int diffY = y - view.y;
    view.updateCoordinates(x,y, diffX, diffY);
    view.setColor(Color.green);
    view.initialize(canvas.getGraphics());
    inPortStatus = false;
    outPortStatus = false;
    portNumber = 3;
    inPortView.initialize();
    outPortView.initialize();
   }
  public void setPtr(PortView port) throws SaosException
  {
    if (port.role == IN)
      model.setInput(port);
    if (port.role == OUT)
      model.setOutput(port);
  }
}
```

```
public class Processor  extends  AObject
{
  public static final int PIdle = 0;
  public static final int PBusy = 1;
  public static final int PComplete = 2;
  public static final int START = 0;
  public static final int STOP = 1;
  public static Random rand = new Random();
  public Queue input;
  public Queue output;
  public AInteger state;
  public int maxProcessingTime = 6;
  public int counter;

  public Processor(String name)
  {
    super(name);
    input = output = null;
    state = new AInteger(PIdle);
    counter = 0;
  }
  public void paint (Graphics g) {}
  public void start () {}
  public void stop () {}
  public void dispatch (int funcIndex) {}
  public void initialize () {}
  public void setInput(PortView port) {
  public void setOutput(PortView port) {
}
```

Appendix B

CODE FOR BEHAVIOR DESCRIPTIONS

This appendix provides the code needed to be filled in the generated classes.

```
class EditGenerator extends EditApp implements Constants
{
  public void initialize() throws SaosException
  {
    .
    .
    .
    for (int i = 0; i < view.objects.size(); i++)
    {
     if (((EditObject) view.objects.elementAt(i)).
                      getClass().getName().equals("EditButton"))
        button = ((EditButton) view.objects.elementAt(i)).button;
     }
    button.selected.addTE((AObject) this, ENABLE,
                               "whenEnabled()", SaosMain.AosUser);
    model.enabled.addTE((AObject) this, DISABLE,
                               "whenDisabled()", SaosMain.AosUser);
                                        // connect EditApp and model
  }
```

```
  public void whenEnabled() throws SaosException
  {
    if (button.selected.val == true)
                                  // button in generator is turned on
      model.enabled.setVal(true);  // enable generator
    else                          // button in generator is turn off
      model.enabled.setVal(false); // disable generator
  }
  public void whenDisabled() throws SaosException
  {
    if (model.enabled.val == false) // generator is disabled when Queue
    {                               // is full, disable generator view
      button.selected.setVal(false);
    }
  }
  public void setPtr(PortView port) throws SaosException
  {
    if (port.role == OUT)
      model.setOutput(port);
  }
}
```

```
public class Generator  extends  AObject
{
  public void initialize () throws SaosException
  {
    state.addTE((AObject) this,
                      START, "start()", SaosMain.AosUser);
    enabled.addTE((AObject) this,
                      START, "start()", SaosMain.AosUser);
    state.addTE((AObject) this,
                      GENERATE, "generate()", SaosMain.AosUser);
                                // initial activation
    FAssign.fAssign((AObject) this, enabled,
                      false, 0, "state", SaosMain.AosUser);
  }
                                // start job generator
  public void start () throws SaosException
  {
    if (enabled.val == true && state.val == GIdle)
    {
      state.setVal(GBusy);
      FAssign.fAssign(this, state, GReady,
                       Math.abs(rand.nextInt() % interval),
                       "state", SaosMain.AosUser);
    }
  }
```

```
                                    // generate a job
  public void generate () throws SaosException
  {
    if (state.val == GReady && output.nJobs.val <
                               output.nSlots)
    {
      if (enabled.val == true)
      {
        counter++;                  // increment job sequence number
        output.nJobs.increment(); // add one job in output queue
      }
      state.setVal(GIdle);
    }
  }

  public void setOutput(PortView port) throws SaosException
  {
    output = (Queue) port.address;
    output.nJobs.addTE((AObject) this, GENERATE,
                       "generate()", SaosMain.AosUser);
  }
}
```

```
class EditQueue extends EditApp implements Constants
{
  public void initialize() throws SaosException
  {
    .
    .
    .
    label.reshape(view.x + (int) (view.w * 0.30),
          view.y + (int) (view.h * 0.25),
                  (int) (view.w * 0.6), view.h / 2);
    canvas.add(label);
    inPortView.address = this.model;
    outPortView.address = this.model;

    model.nJobs.addTE((AObject) this,
                  UPDATE, "ifpdated()", SaosMain.AosUser);
  }

  public void ifupdated() throws SaosException
  {
    label.hide();
    label.setText("Job= " + model.nJobs.val);
    label.show();
    if (SaosMain.debugLevel.val > 3)
      System.out.println("queue updated " + model.nJobs.val);
  }
}
```

```
class  EditProcessor  extends  EditApp implements Constants
{
  public void initialize() throws SaosException
  {
    .
    .
    .
    try
    {
      model.state.addTE((AObject) this,
                  PIdle, "ifavail()", SaosMain.AosUser);
    } catch (SaosException er)
    {
      System.out.println("ERROR: " + er);
    }
  }

  public void ifavail()
  {
    if (model.state.val == PIdle)
      view.setColor(Color.green);
    else
      view.setColor(Color.red);
    view.paint(this.canvas.getGraphics());
  }

  public void setPtr(PortView port) throws SaosException
  {
    if (port.role == IN)
      model.setInput(port);
    if (port.role == OUT)
      model.setOutput(port);
  }
}
```

```
public class Processor  extends  AObject
{
  public void initialize () throws SaosException
  {
    state.addTE((AObject) this, START, "start()", SaosMain.AosUser);
    state.addTE((AObject) this, STOP, "stop()", SaosMain.AosUser);
  }
                                // start job processing
  public void start () throws SaosException
  {
    // start processing if input has jobs and processor is free
    if (input.nJobs.val > 0 && state.val == PIdle)
    {                           //remove a job from input queue
      input.nJobs.decrement();
      counter++;
      state.setVal(PBusy);      // make processor busy

    // sets state to PComplete after processing completes
      FAssign.fAssign((AObject) this, state, PComplete,
                Math.abs(rand.nextInt()
                % maxProcessingTime), "state", SaosMain.AosUser);
    }
  }
  public void stop () throws SaosException
  {
    if (output != null)
      if (state.val == PComplete && output.nJobs.val < output.nSlots)
      {
        output.nJobs.increment();
        if (SaosMain.debugLevel.val > 1)
        state.setVal(PIdle);
      }
  }
```

```
  public void setInput(PortView port) throws SaosException
  {
    input = (Queue) port.address;
    input.nJobs.addTE((AObject) this, START, "start()",
            SaosMain.AosUser);
  }
  public void setOutput(PortView port) throws SaosException
  {
    output = (Queue) port.address;
    output.nJobs.addTE((AObject) this, STOP, "stop()",
            SaosMain.AosUser);
  }
}
```

Appendix C

CODE FOR CONNECTING ENTITIES

This appendix provides the code for class `Link`, and class `PortView`. These classes are essential for connecting entities together.

```
class Link extends SchemaRelationship implements Constants
{
  PortView firstPortView;
  PortView secondPortView;
  int  fromObjectID = -1;        // from objectID of an EditApp
  int  toObjectID = -1;          // to objectID of an EditApp

  public Link(Color color, int x1, int y1, int x2, int y2,
              SchemaCanvas canvas, int direction, int cardinal)
  {
    super(color, x1, y1, x2, y2, canvas, direction, cardinal);
  }
  public boolean connectPorts(PortView portView1, PortView portView2,
              SchemaRelationship arrow, AppCanvas canvas)
  {
    boolean connected = false;
    if (connected == false) return false;
    // link an active object to other PortView
    firstPortView.link(secondPortView);
    // link that PortView to its EditApp's model
    firstPortView.editApp.setPtr(firstPortView);
    return true;
  }
}
```

```
class PortView extends EditFillRectangle implements Constants
{ int           role;
  int           multiplicity;
  AObject       address;
  Vector        objects;
  public PortView(int role, AppCanvas canvas,
                  EditApp editApp, int multiplicity)
  { this.role    = role;
    this.multiplicity = multiplicity;
    this.editApp = editApp;
  }
    // connect this (first click) to another portView (second click)
  public Link connectPorts(PortView portView)
  { // find objectID of EditApp that a PortView is connected/created
    String thisKind = this.getObjectKind();
    String anotherKind = portView.getObjectKind();
    // get information about direction and cardinality from EERD
    SchemaRelationship arrow = findSchemaRelationshipByKind(
         canvas.erCanvas.objects, thisKind, anotherKind);
    if (arrow != null) {   // create a Link
      Link link = new Link(arrow.color, this.x + this.w / 2,
            this.y + this.h / 2, portView.x + portView.w / 2,
            portView.y + portView.h / 2,
            arrow.canvas, NO_DIRECTION, ONE_ONE);
      boolean success =    // connect two PortViews
            link.connectPorts(this, portView, arrow, canvas);
      if (success == true) // a Link is successfully created
      { link.initialize(canvas);
                           // increment number of Ports connected
        this.numberOfPortConnected += 1;
        portView.numberOfPortConnected += 1;
        this.linkVector.addElement((Object) link);
        portView.linkVector.addElement((Object) link);
        return link;
      }
    } else
        return null;
  }
}
```

Appendix D

EXPERIMENTS

This appendix provides descriptions of Experiment $1A$, $1B$, $2A$, and $2B$. Experiment $1A$ investigates comprehensibility of OMT class diagrams, $1B$ comprehensibility of EERD design documents, $2A$ software composition with a menu-based saos editor and $2B$ software composition with the entity-relationship software development environment (ERSDE).

D.1 Experiment $1A$: Comprehensibility of OMT Class Diagrams

D.1.1 Definitions of OMT Symbols

Fig. D.1 shows some symbols used in class diagrams of Object Modeling Technique (OMT). A *class* is represented by a rectangle, and an *association type* by a link. A *cardinality ratio* for an association type is shown with a small filled-circle placed on a *many-side* of mapping.

FIGURE D.1: Symbols used by Object Modeling Technique.

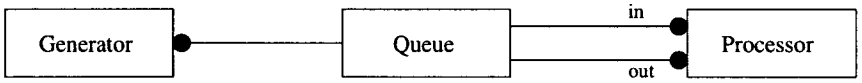

FIGURE D.2: OMT class diagram for queueing systems.

D.1.2 Example of an OMT Class Diagram

Fig. D.2 shows an OMT class diagram for queueing systems. There are three classes: `Generator`, `Queue`, and `Processor`. Each `Generator` produces jobs and stores them in a `Queue`. A `Queue` holds jobs. Each `Processor` takes out a job from a `Queue`, processes it, and passes the processed job to another `Queue`.

D.1.3 A Queueing System Application

We now can compose a queueing system application as shown in Fig. D.3 by using the OMT class diagram in Fig. D.2 as a template. The application is composed from three generators, five queues, and five processors. Generator g1 is connected to queue q1, and generators g2 and g3 to queue q2. Queue q1 is connected to processors p1 and p2, and queue q2 to processor p3. Processor p1 is connected to queue q3, and so on.

The cardinality-ratio constraint associated with each relationship type is observed as follows. Multiple (0 through many) generators are connected to each queue, but each generator is connected to at most one queue. Multiple processors retrieve jobs from one queue, and multiple processors send jobs to one queue. However, each processor retrieves jobs from at most one queue, and it sends jobs to at most one queue.

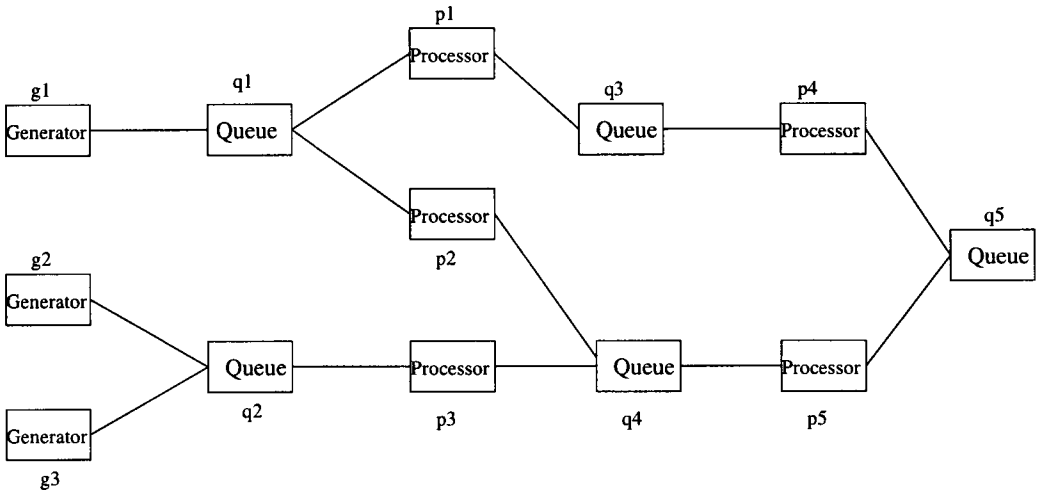

FIGURE D.3: A queueing system application conforming to the OMT class diagram.

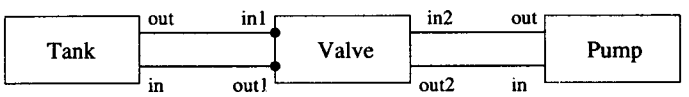

FIGURE D.4: An OMT class diagram for tank systems.

D.1.4 Problem I.

Step I. Please study and understand the OMT class diagram shown in Fig. D.4. Then, compose an application that utilizes this diagram as a template as requested in Step II.

Step II. Please compose a tank system application by using the previous class diagram as a template. There should be at least five tanks, eight valves, and four pumps. You can have more components if needed. Show as many variant configurations as

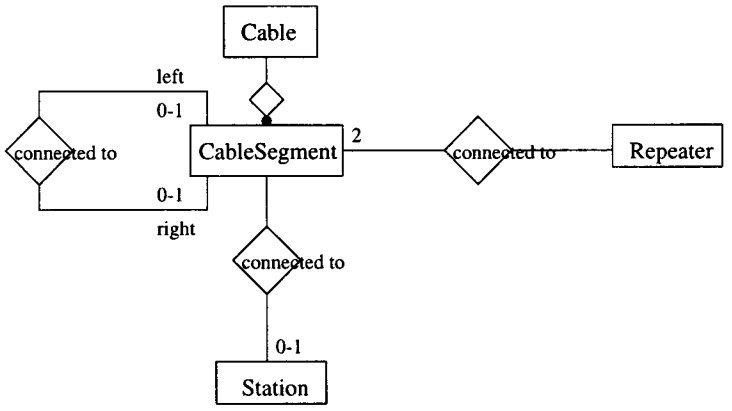

FIGURE D.5: An OMT class diagram for local area network systems.

possible. You may refer to the class diagram on the previous page while composing your application.

D.1.5 Problem II.

Step I. Please study and understand the diagram in Fig. D.5. You will be asked to compose an application that utilizes this diagram as a template as requested in Step II.

Step II. Please compose an application. There should be at least three cables, two repeaters, and two stations. Each cable should have at least five cable segments. You can have more components if needed. Show as many variant configurations as possible. You may refer to the class diagram on the previous page while composing your application.

D.1.6 Problem III.

Fig. D.6 shows an OMT class diagram for tank systems, and Fig. D.7 an extended entity-relationship diagram (EERD) for the same application domain. In fact these two diagrams represent the same information; i.e., they are semantically identical.

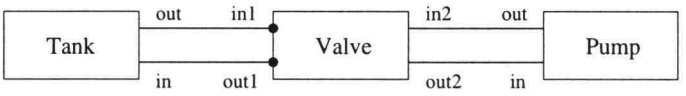

FIGURE D.6: An OMT class diagram for tank systems.

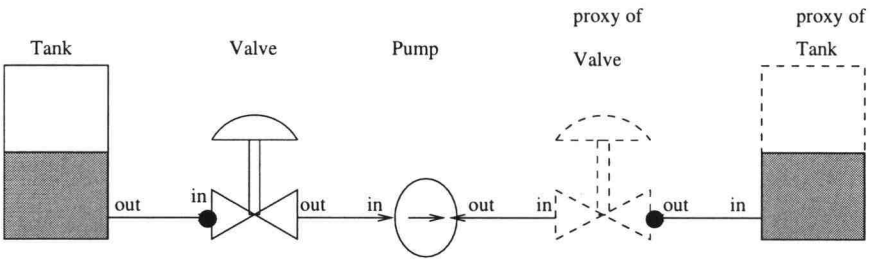

FIGURE D.7: An EERD for tank systems.

Which one of these two diagrams do you think is better? Please explain why do you think so.

D.1.7 Problem IV.

Fig. D.8 shows an OMT class diagram for local area network systems, and Fig. D.9 an extended entity-relationship diagram (EERD) for the same application domain. In fact these two diagrams represent the same information; i.e., they are semantically identical. Which one of these two diagrams do you think is better? Please explain why do you think so.

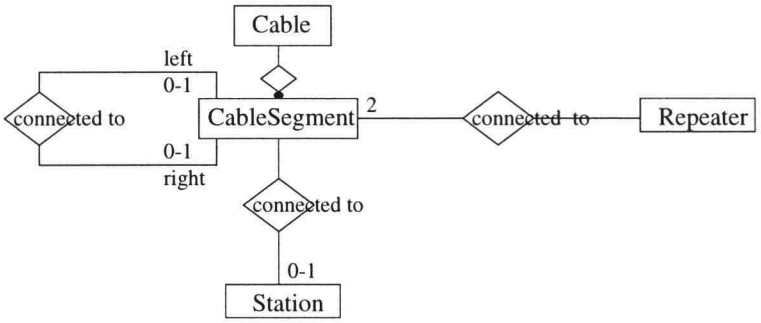

FIGURE D.8: An OMT for local area network systems.

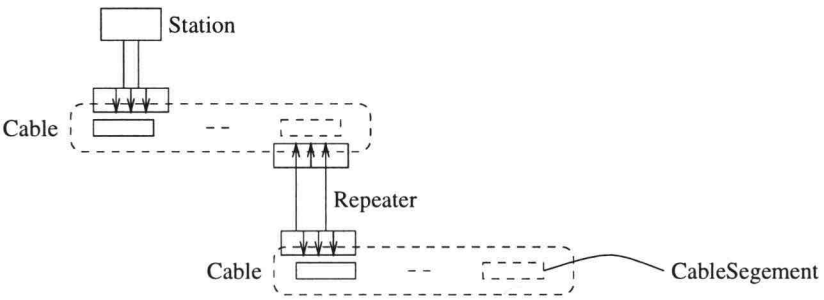

FIGURE D.9: An EERD for local area network systems.

D.2 Experiment $1B$: Comprehensibility of EERD Design Documents

D.2.1 Definitions of Extended Entity-Relationship Diagrams (EERDs) Notations

Fig. D.10 shows some symbols used in EERDs. An entity type is normally represented by a rectangle, but it can be replaced by its iconic representation. A link is an arrow which indicates the direction of data access. A cardinality ratio is represented by a small filled-circle placed on a *many-side* of mapping. A *proxy entity type*, which is drawn with dashed lines, is equivalent to its original entity type.

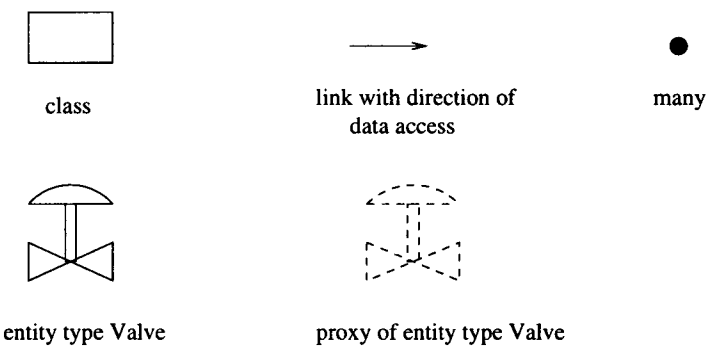

FIGURE D.10: Symbols used by the extended entity-relationship diagrams.

D.2.2 Example of an EERD

Fig. D.11 is an EERD for a queueing system. There are three entity types: `Generator`, `Queue`, and `Processor`. A proxy of entity type `Queue` is also used. Each `Generator` produces jobs and stores them in a `Queue`. A `Queue` holds jobs. Each `Processor` takes a job from a `Queue`, processes it, and passes the processed job to another `Queue`.

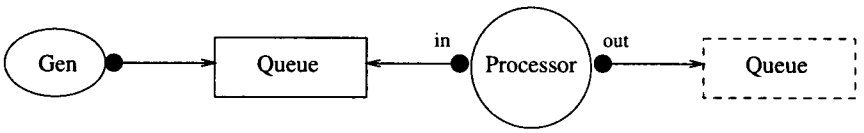

FIGURE D.11: An EERD for queueing systems

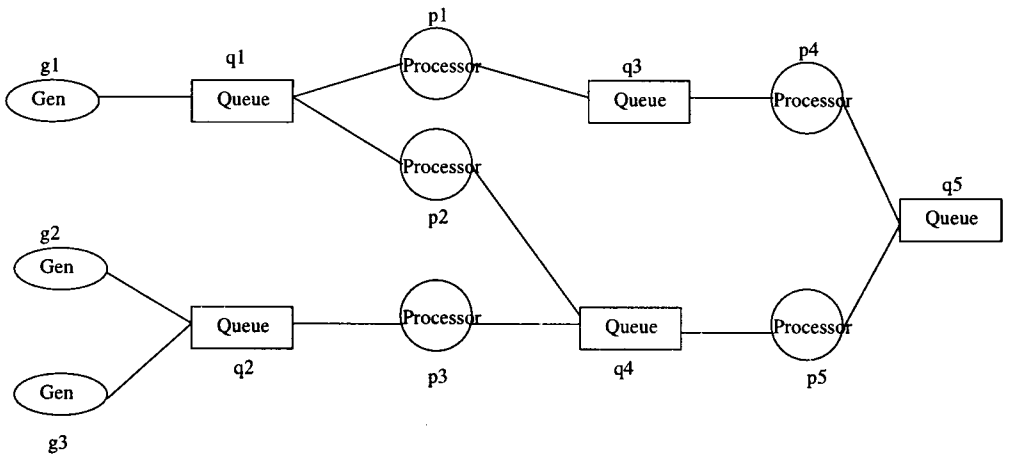

FIGURE D.12: A queueing system application conforming to the EERD.

D.2.3 A Queueing System Application

We now can compose an application of a queueing system as shown in Fig. D.12 by using the EERD in Fig. D.11 as a template. The application is composed from three generators, five queues, and five processors. Generator g1 is connected to queue q1, and generators g2 and g3 to queue q2. Queue q1 is connected to processors p1 and p2, and queue q2 to processor p3. Processor p1 is connected to queue q3, and so on.

The cardinality-ratio constraint associated with each relationship type is observed as follows. Multiple (0 through many) generators are connected to each queue, but each generator is connected to at most one queue. Multiple processors retrieve jobs

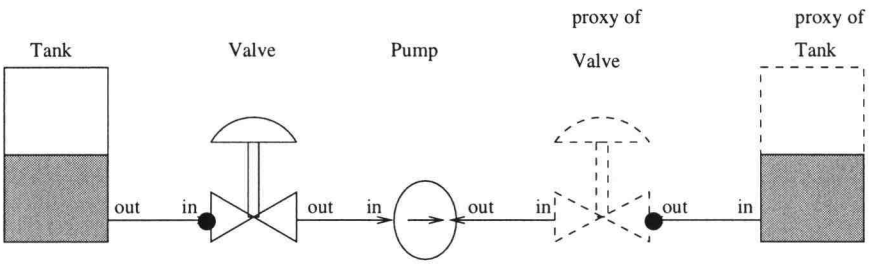

FIGURE D.13: An EERD for tank systems.

from one queue, and multiple processors send jobs to one queue. However, each processor retrieves jobs from at most one queue, and it sends jobs to at most one queue.

D.2.4 Problem I.

Step I. Please study and understand the EERD shown in Fig. D.13. Then, compose an application that utilizes this diagram as a template as requested in Step II.

Step II. Please compose an application by using the previous class diagram as a template. There should be at least five tanks, eight valves, and four pumps. You can have more components if needed. Show as many variant configurations as possible. You may refer to the EERD on the previous page while composing your application.

D.2.5 Problem II.

Step I. Please study and understand the diagram in Fig. D.14. You will be asked to compose an application that utilizes this diagram as a template as requested in Step II.

Step II. Please compose an application by using the previous class diagram as a template. There should be at least three cables, two repeaters, and two stations. Each cable should have at least five cable segments. You can have more components

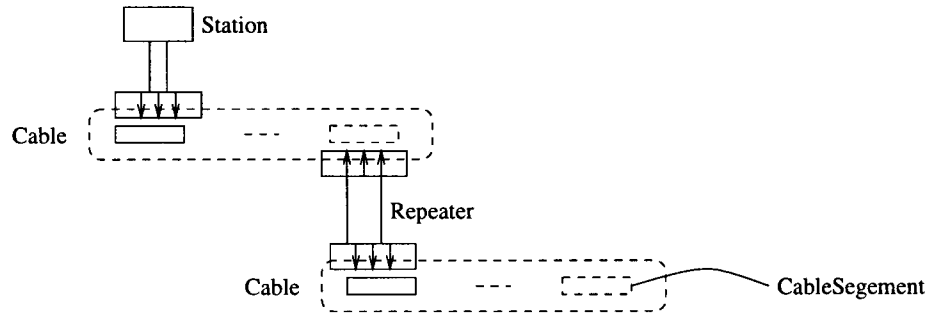

FIGURE D.14: An EERD for local area network systems.

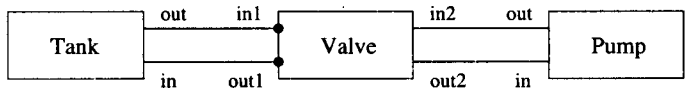

FIGURE D.15: An OMT class diagram for tank systems.

if needed. Show as many variant configurations as possible. You may refer to the EERD on the previous page while composing your application.

D.2.6 Problem III.

Fig. D.15 shows an OMT class diagram for tank systems, and Fig. D.16 an extended entity-relationship diagram (EERD) for the same application domain. In fact these two diagrams represent the same information; i.e., they are semantically identical. Which one of these two diagrams do you think is better? Please explain why do you think so.

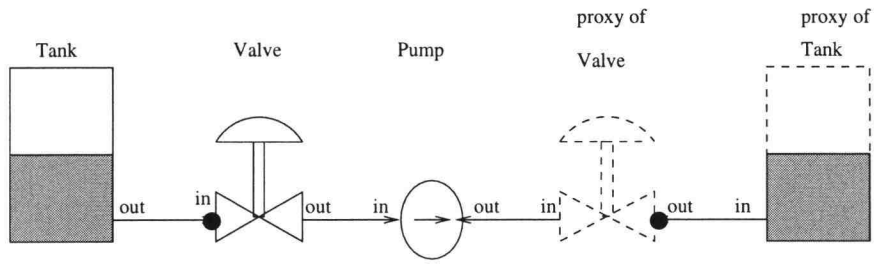

FIGURE D.16: An EERD for tank systems.

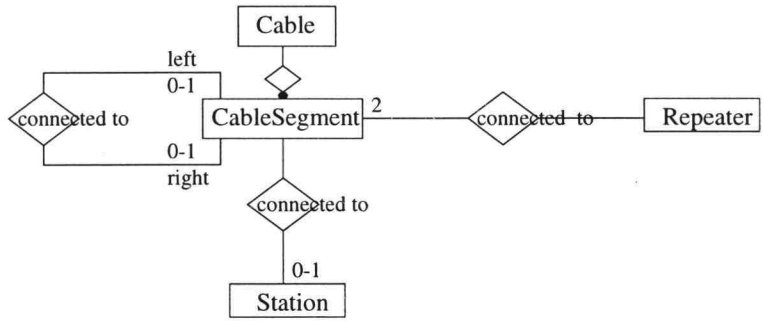

FIGURE D.17: An OMT for local area network systems.

D.2.7 Problem IV.

Fig. D.17 shows an OMT class diagram for local area network systems, and Fig. D.18 an extended entity-relationship diagram (EERD) for the same application domain. In fact these two diagrams represent the same information; i.e., they are semantically identical. Which one of these two diagrams do you think is better? Please explain why do you think so.

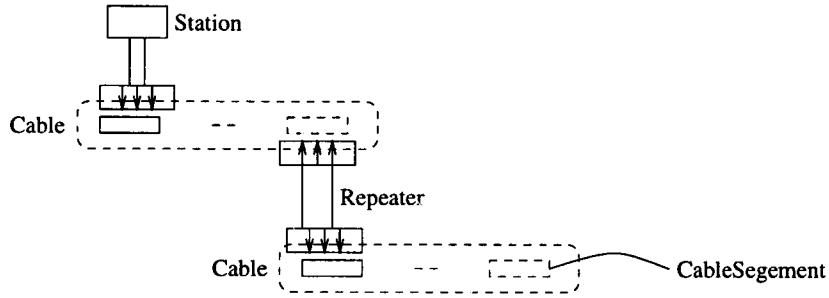

FIGURE D.18: An EERD for local area network systems.

D.3 Experiment 2A: Software Composition with a Menu-Based SAOS Editor

D.3.1 Composing a Tank-System Application

We first demonstrate how to compose a tank-system simulator by using a menu-based structural active-object system (SAOS) editor shown in Fig. D.19. The upper canvas area of this editor shows the menu items for the available component types, and the lower canvas area displays a tank-system application being constructed.

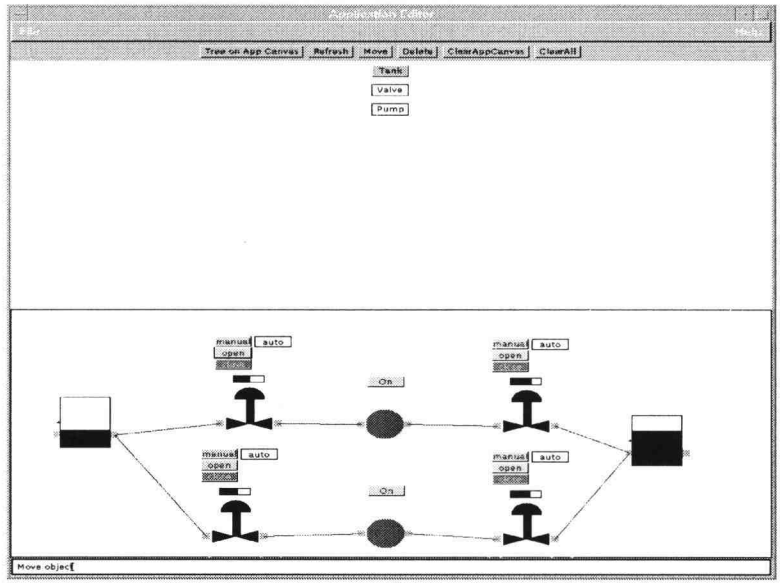

FIGURE D.19: A menu-based SAOS application editor for a tank system application.

D.3.2 Description of a Tank System

A tank system consists of tanks, valves, and pumps. A tank contains liquid, a pump makes liquid flow, and a valve controls the amount of flow. The liquid flows from left to right. The output end of a tank can be connected to the input ends of possibly

multiple valves, and the output end of a valve can be connected to the input end of either a pump or a tank. The output end of a pump can be connected to the input end of at most one valve, and the input end of a tank can be connected to the output ends of possibly multiple valves.

D.3.3 An Example of a Tank-System Simulator Application

The tank system application shown in Fig. D.19 is composed from two `Tanks`, four `Valves`, and two `Pumps`. The output of the first `Tank` is connected to the input-port view of the two `Valves` in the left half of the canvas area, and the input of the second `Tank` is connected to the output-port view of the two `Valves` in the right half of the canvas area. Each `Pump` is connected to the `Valves` on its left and right.

D.3.4 Creating a Component

The user can instantiate a component of an entity type by clicking a left mouse button on the menu-item button for that entity type and then by clicking the left mouse button again inside the lower canvas area. The component is created and placed at the location where the second mouse click occurs.

D.3.5 Connecting Components

Once components are created, they can be connected with each other. Each component has two port views for this purpose: the input-port view on its left and the output-port view on its right. We can connect two components by clicking the left mouse button once inside the output-port view of one component and again inside the input-port view of the other component. Components are active as soon as they are created, and the user can change their states by pressing buttons. In this example both `Pumps` are on, and the manual open-operation is selected for all the `Valves`.

D.3.6 Task I: Composing a Queueing System Application

We have a menu-based SAOS application editor for queueing systems as shown in Fig. D.20. Now, construct a queueing system application. The application should have at least two `Generators`, at least four `Queues`, and at least four `Processors`. You can have more components if needed. Show as many variant configurations as possible. You may consult an online Help menu for more information on how to instantiate and connect entities. Please give us any comments on the usability of this editor.

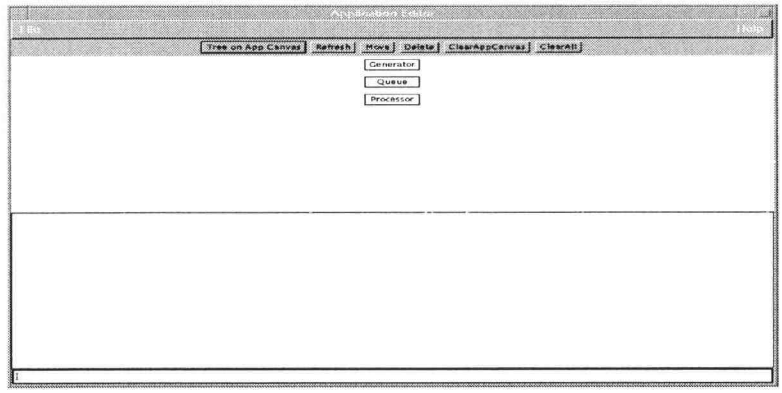

FIGURE D.20: A menu-based SAOS application editor for a queueing system application.

D.3.7 Task II: Composing a Local Area Network Application

We have a menu-based SAOS application editor for local area network systems as shown in Fig. D.21. Now, construct a local area network system application. The application should have at least three `Cables`, two `Repeaters`, and two `Stations`. Each cable should have at least five cable segments. You can have more components if needed. Show as many variant configurations as possible. You may consult an

online Help menu for more information on how to instantiate and connect entities. Please give us any comments on the usability of this editor.

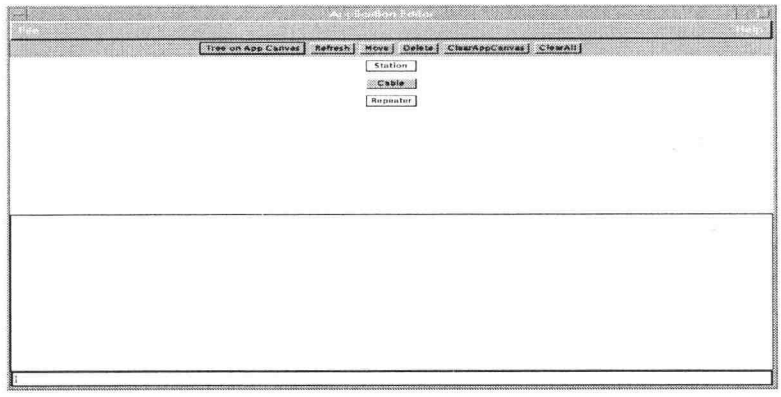

FIGURE D.21: A menu-based SAOS application editor for a local area network system application.

D.4 Experiment 2*B*: Software Composition with Entity-Relationship Software Development Environment (ERSDE)

D.4.1 Composing a Tank-System Application

We first demonstrate how to compose a tank-system simulator by using the entity-relationship software development environment (ERSDE) shown in Fig. D.22. This editor displays the extended entity-relationship diagram (EERD) for tank systems in the upper canvas area so that it can be used as a template for a tank system application to be constructed, and the lower canvas area displays the tank-system system application being constructed.

D.4.2 An EERD of a Tank System

An EERD specifies what kinds of entities can be created and how the entities can be connected to another one. The EERD for tank systems contains three entity types `Tank`, `Valve`, and `Pump` and two proxy entity types for (`Valve` and `Tank`). A proxy entity type displayed in light gray is equivalent to the original entity type. Each entity has two port views: the input-port view on its left and the output-port view on its right.

The EERD for tank systems tells the following rules for the possible connections among `Tanks`, `Valves`, and `Pumps`. The output-port view of a `Tank` can be connected to the input-port view of possibly multiple `Valves`, since the relationship type between the entity type `Tank` and the entity type `Valve` is one-to-many. The output-port view of a `Valve` can be connect to the input-port view only at most one `Pump`. The output-port view of a `Pump` can be connected to the input-port view of at most one `Valve`. The output-port view of each `Valve` can be connected to the input-port view of only one `Tank`, and the input-port view of a `Tank` can be connected to the output-port views of possibly multiple `Valves`.

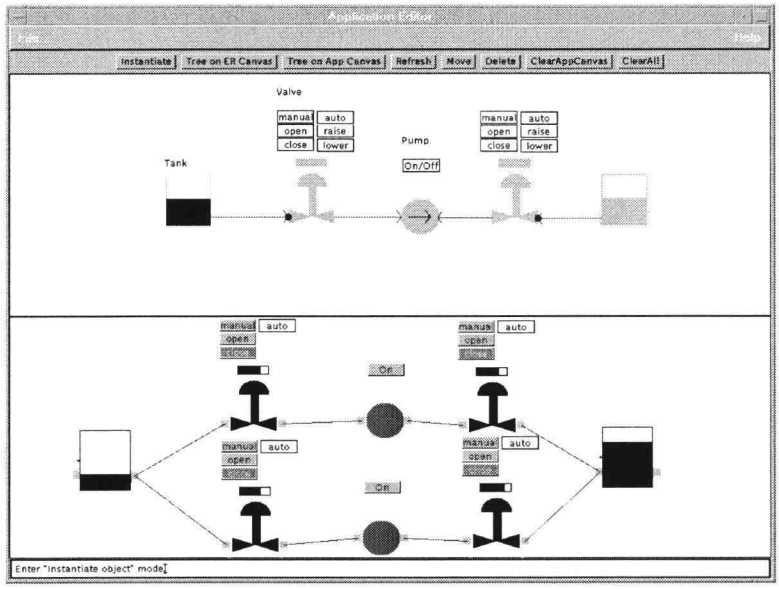

FIGURE D.22: An EERD and an application of a tank system

D.4.3 An Example of a Tank-System Application

The tank system application shown in Fig. D.22 is composed from two `Tanks`, four`Valves`, and two `Pumps`. The output-port view of the first `Tank` is connected to the input-port views of the two `Valves` in the left half of the canvas area, and the input-port view of the second `Tank` is connected to the output-port views of the two `Valves` in the right half of the canvas area. Each `Pump` is connected to the `Valves` on its left and right.

D.4.4 Instantiating an Entity

To create an instance of an entity type, the user must first select the instance-creation mode by clicking a left mouse button on the `Instantiate` button. Then select on the EERD the entity type whose instance should be created by clicking the left mouse

button on it. Now each time the left mouse button is clicked inside the lower canvas area, an instance of the selected entity type is created and placed at the location where the mouse click occurs.

D.4.5 Connecting Entities

Once entities are created, the user can connect with each other. Each entity has two port views for this purpose: the input-port view on its left and the output-port view on its right. We can connect two entities, when the editor is in an `Instantiate` mode, by clicking the left mouse button once inside the output-port view of one entity and again inside the input-port view of the other entity. Entities are active as soon as they are created, and the user can change their states by pressing buttons. In this example both `Pumps` are on, and the manual open-operation is selected for all the `Valves`.

D.4.6 Task I: Composing a Queueing System Application

We have an ERSDE application editor for queueing systems as shown in Fig. D.23. Now, construct a queueing system application. The application should have at least two generators, at least four queues, and at least four processors. You can have more components if needed. Show as many variant configurations as possible. You may consult an online Help menu for more information on how to instantiate and connect entities and execute an application. Please give us any comments on the usability of this editor.

D.4.7 Task II: Composing a Local Area Network Application

We have an ERSDE application editor for local area network systems as shown in Fig. D.24. Now, construct a local area network system application. The application should have at least three cables, two repeaters, and two stations. Each cable should have at least five cable segments. You can have more components if needed. Show

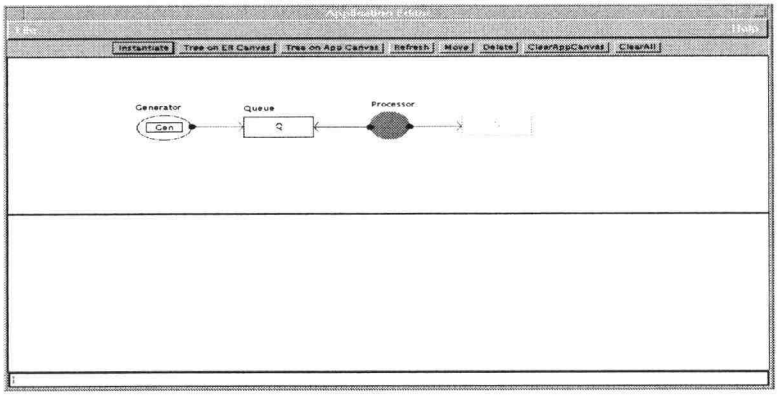

FIGURE D.23: An ERSDE application editor for queueing systems.

as many variant configurations as possible. You may consult an online Help menu for more information on how to instantiate and connect entities and execute an application. Please give us any comments on the usability of this editor.

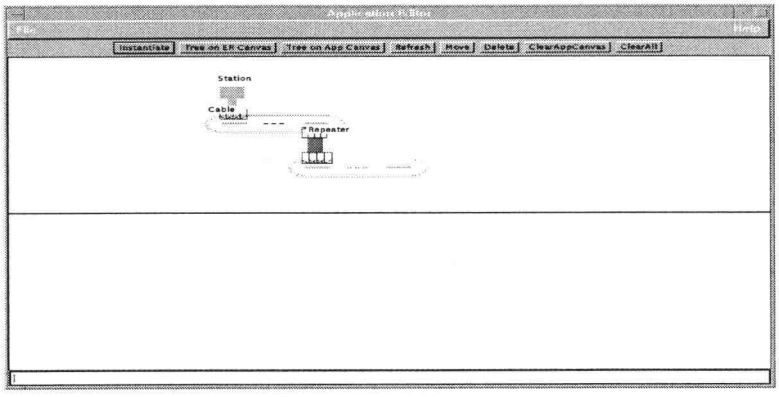

FIGURE D.24: An ERSDE application editor for local area network systems.

Appendix E

STATISTICAL TEST METHODS

In this appendix, we explain the three kinds of statistical test methods used in this research. They are *Fisher's exact tests*, *One-sample sign tests*, and *Mann-Whitney two-samples tests* [22, 67].

E.1 Fisher's Exact Test Method

We used the Fisher's exact test method to test whether or not the proportion of the number of the students who obtained correct answers by using EERDs is statistically greater than that of the number of the students who obtained correct answers by using OMT class diagrams. The null hypothesis is the proportion of the number of the students who obtained correct answers by using EERDs is less than or equal to that of the number of the students who obtained correct answers by using OMT class diagrams.

The $p-value$ is the probability of obtaining a statistic as extreme or more extreme than the statistic in its evidence against the null hypothesis, if the null hypothesis is correct [69]. That is, the one-tailed $p - value$ of Fisher's exact test method is the cumulative probability in Equation. E.1 of the observed a or less for the left tail, and the observed a or more for the right tail [67], where a is the observed value in Table E.1. We reject the null hypothesis when the $p - value$ is less than or equal to the significant level $\alpha = 0.05$.

	EERDs	*OMT*	Total
# of Students got Correct Answers	a	b	$a+b$
# of Students got Incorrect Answers	$A-a$	$B-b$	$A+B$
Total # of Students	A	B	N

TABLE E.1: Composition results.

The conditional distribution of A given $a+b$ is the hypergeometric distribution, with discrete frequency function given by

$$f(a\backslash A, B, a+b) = \frac{\binom{A}{a}\binom{B}{b}}{\binom{N}{a+b}} \tag{E.1}$$

For example, if we apply the equation to the statistic given in Table. E.2. That is $a = 7, b = 1, A = 9$, and $B = 8$. We get the $p - value$ of 0.01 with $a >= 7$. With this $p - value$, which is less than 0.05, we can conclude that the proportion of the number of the students who obtained correct answers by using EERDs is statistically greater than to that of the number of the students who obtained correct answers by using OMT class diagrams.

	EERD	OMT	Total
# of Students got Correct Answers	7	1	8
# of Students got Incorrect Answers	2	7	9
Total	9	8	17

TABLE E.2: Composition of local-area network systems by CS361 students.

E.2 One-Sample Sign Test Method

We used a one-sample sign test method to test whether or not the subjects preferred EERDs statistically more than OMT class diagrams. The null hypothesis is the subjects did not prefer EERDs statistically more than OMT class diagrams.

The $p - value$ for this one-sample sign test method is the cumulative probability of the observed value of a or less than for the left tail, and the observed a or greater than for the right tail. The $p - value$ can also be obtained from Table A.1 of Pratt [67]. We reject the null hypothesis when the $p - value$ is less than or equal to the significant level $\alpha = 0.05$.

For example, the data in Table. E.3 indicates that the number of the students who preferred the EERD for tank systems is 12, while the number of those who preferred the OMT class diagram for the same application domain is 2. With Table A.1 of Pratt, we found that the $P(K <= 2/14, 0.50) = 0.0066$, which is the $p-value$. With this $p-value$, we conclude that the significant number of students statistically preferred the EERDs to the OMT class diagrams for the application domains of tank systems and local-area network systems.

Methods	OMT	EERD
# of subjects	2 (a)	12
%	14	86

TABLE E.3: Preference of the diagramming methods for tank system application by the CS361 students.

E.3 Mann-Whitney Two-Samples Test Method

We used the Mann-Whitney two-samples test method to test whether or not the average time taken by the subjects using the entity relationship software development environment (ERSDE) application editor to compose the application is statistically less than those using the menu-based structural active-object system (SAOS) graphical editor. The null hypothesis is the average time taken by the subjects using the entity relationship software development environment (ERSDE) application editor to compose the application is greater than or equal to those using the menu-based structural active-object system (SAOS) graphical editor.

We compute the test statistic as follows.

1. Combine the two samples.

2. Rank all sample observations from the smallest to the largest.

3. Compute the observed values of the test statistic,

   The test statistic is

   $$T = S - \frac{n_1(n_1+1)}{2} \tag{E.2}$$

   where S is the sum of ranks assigned to the sample observations from population I.

4. We reject the null hypothesis if the T-statistic is less than $W_{0.05,n_1,n_2}$, where the value of $W_{0.05,n_1,n_2}$ is provided in Table A.8 of Daniel [22].

From the data in Table. E.4, we get $S = 109.5$, and hence the test statistic $T = 16$. With Table A.8 of Daniel, we found that $W_{0.05,9,9} = 22$, $W_{0.95,9,9} = 59$, and $p - value = 0.01$. With this $p - value$ less than 0.05, we conclude that the subjects using the ERSDE application editor took statistically less time than those using menu-based SAOS graphical editors.

SAOS	Rank	ERSDE	Rank
5	4	2	1
6	6.5	4	2.5
6	6.5	4	2.5
9	12.5	6	6.5
10	14.5	6	6.5
10	14.5	7	9
11	17	8	10.5
11	17	8	10.5
11	17	9	12.5
8.78 (Average)	109.5 Total	6.00 (Average)	61.5 Total

TABLE E.4: Times (in minutes) used to compose queueing system applications.