Order Number 9506208

HAT (Hyper Analysis Toolkit): A tool for hypertext-based dynamic systems analysis

He, Jingxiang, Ph.D.

University of Hawaii, 1994

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

HAT (HYPER ANALYSIS TOOLKIT): A TOOL FOR HYPERTEXT-BASED DYNAMIC SYSTEMS ANALYSIS

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMMUNICATION AND INFORMATION SCIENCES

AUGUST 1994

By

JingXiang He

Dissertation Committee:

Kenneth A. Griggs, Chairperson
William E. Remus
Rosemary H. Wild
Larry N. Osborne
David C. H. Yang

ACKNOWLEDGMENTS

I am profoundly grateful to Dr. Kenneth Griggs for his advice and support throughout this research. I am equally grateful to all other members of my committee, Dr. William Remus, Dr. Rosemary Wild, Dr. Larry Osborne, and Dr. David Yang. Each of them took time out from their busy schedules to help my progress. My special thanks to Dr. Art Lew for editing and evaluating this dissertation.

Many thanks to Mr. Tien Lum for his participation in this project. I also own a debt of gratitude to Mr. Jon Fujiwara and my colleagues at the Computer Resources of the College of Business Administration for their understanding and technical support. My thanks also go to Dr. Timothy Hill and my friends in the CIS program for providing information that makes this dissertation possible.

Last but not least, my special thanks to my wife XiaoMei for her love and emotional support throughout the four-year endeavor of my Ph.D. I also want to express gratitude to my parents in China for their continuous care and love.

ABSTRACT

Increasing system complexity necessitated the development of software engineering methods and CASE (Computer Aided Software Engineering) tools. Many software developers and businesses have adopted engineering principles and computer aided tools to cope with the growing needs of software development and maintenance. In practice, most software projects are initiated by the information needs of the end users. Precise descriptions and understandings of these information needs are critical to information systems. It is believed that increasing end user involvement and doing things right in the early stages of software development processes are the most effective ways to improve software quality.

This dissertation presents a research project to develop a tool, HAT (Hyper Analysis Toolkit), to help the end users to understand and use the structured analysis techniques. HAT provides a hypertext linkage of graphical models, such as DFDs (Data Flow Diagrams) and ERDs (Entity Relation Diagrams), with system description narratives and other documents created during the system analysis. Hyperlinks placed in the diagrams and documents provide an easy way for end users and system analysts to navigate and cross-reference the system models.

Model evaluation is as important as model description. In addition to the hypertext-based user interface for model description, this research incorporates a simulation package and a rule-based expert system to estimate the dynamic features of a DFD model. Dynamic evaluation of models at early stages will help system developers and end users to have better control over software development processes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CASE	Computer Aided Software Engineering
DDE	Dynamic Data Exchange
DDEML	DDE Management Library
DFD	Data Flow Diagram
DIF	Document Integration Facility
DLL	Dynamic Link Library
DSS	Decision Support Systems
E-JAD	Electronic JAD
ERD	Entity Relation Diagram
EMS	Electronic Meeting System
ESS	Expert Simulation System
FIFO	First-in-first-out
4GL	4th Generation Language
GDI	Graphic Device Interface
GSS	Group Supporting System
GUI	Graphical User Interface
HAM	Hypertext Abstract Machine
IDE	Integrated Development Environment
IS	Information System
ISHYS	Intelligent Software HYpertext System
JAD	Joint Application Design
MDI	Multiple Document Interface
MFC	Microsoft Fundamental Classes
NEST	NEtwork Simulation Testbed
OLE	Object Linking and Embedding
OWL	Object Window Library
RAD	Rapid Application Development
SA	Systems Analysis

SD	Systems Design
SDLC	System Development Life Cycle
SE	Software Engineering
SESSA	Statistical Expert System for Simulation Analysis
SILK	Speech, Image, Language and Knowledge
SMDE	Simulation Model Development Environment
SGML	Standard Generalized Markup Language
SPG	Simulation Program Generator
UIDS	User Interface Development Systems
UIT	User Interface Toolkits
VBX	Visual Basic eXtension
VIM	Visual Interactive Modeling
VIS	Visual Interactive Simulation
WAF	Windows Application Framework
WIMP	Window, Icon, Menu and Point

CHAPTER 1 INTRODUCTION

The development of contemporary computer technology enables computer applications in many areas and places increasing pressure on information system professionals and end users. Because of the increased information system requirements, more information systems need to be developed, maintained and upgraded. It is paramount that computer scientists and information system developers deal with these increasing demands.

This research addresses the problem of user requirement specifications and information system (IS) modeling at the systems analysis (SA) stage. Systems analysis is the first stage in the software development life cycle (SDLC) in which users meet system analysts. Requirement specifications for a target system are defined at this stage. The cost of software modification increases exponentially as the development process goes from the SA stage to later stages of the SDLC [Sommerville 89]. Since changes can be least expensively made at the SA stage, it is obvious that different alternatives should be carefully weighted to determine the most feasible solution at this stage. This thesis reports on a project that incorporates hypertext, simulation and expert system techniques to construct a tool – Hyper Analysis Toolkit (HAT). The essence of this project is twofold: (1) to improve the communication among users and system analysts by introducing a hypertext-based user interface, and (2) to enhance model evaluation by integrating simulation and expert system techniques. This thesis reviews what has been done in related areas and describes a unique architecture feasible for the implementation of HAT. This architecture may also be used for visual interactive simulation environment and other simulation studies.

Several different fields are involved in this research. The first chapter begins with some background information on software engineering and computer-aided software engineering (CASE) tools. It answers the questions of 'what' and 'why' of this research. The second chapter reviews what has been done on this and related subjects. The third chapter summarizes the methodologies and tools used for this project, as well as some theories behind them. The fourth chapter focuses on the description of the HAT architecture and systems design. The fifth chapter discusses some implementation issues and lessens learned from this project. The thesis concludes with the sixth chapter that summarizes the strengths, weaknesses, contributions, as well as possible extensions of this research.

1.1 Software Engineering and Software Life Cycle

The term 'software engineering' (SE) was first introduced in the late 1960's at a conference held to discuss the 'software crisis'[Sommerville 89]. The software crisis resulted from the introduction of third generation computers. The advanced hardware technologies and powerful computers made the software methodologies inadequate to meet the increasing needs for more and larger applications. After more than twenty years, the software crisis is still with us. The demand for software has increased at a faster rate than the improvement in the productivity of software engineers. Furthermore, the advent of microcomputer systems has increased the awareness of computer applications and brought more people into computer-related activities. However, some of these people are not aware of SE methodologies and are repeating the same mistakes made by software engineers twenty years ago. Currently, there is a great need for better tools, techniques, methodologies, and most importantly, better education and training for IS developers as well as end users.

Although the definition of software engineering varies, the common factors are that (1) SE is concerned with software systems which are built by teams rather than by individual programmers, (2) SE uses engineering principles in the development of information systems, and (3) SE consists of both technical and non-technical aspects. A well-engineered software project should have the following features as described by Sommerville [Sommerville 89]:

(1) *The software should be maintainable:* As long-life software is subject to regular change, it is important that the software is written and documented in such a way that changes can be made without undue cost.

(2) *The software should be reliable:* An appropriate level of reliability is essential if a software system is to be of any use.

(3) *The software should be effective and efficient:* The software should perform defined functions with a minimum cost of time and computer resources.

(4) *The software should offer an appropriate user interface:* The software cannot be used to its full potential if a user interface makes it difficult to use. The user interface design should take into account the capacity and background of the intended users.

SE studies have developed a number of general models of software development. Table 1.1 shows a list of some general models. Among these models, the 'waterfall' model, which was proposed by Royce in 1970[Royce 70], is the earliest and most frequently used model for IS development. It divides the software development life cycle (SDLC) into five different stages: (1) *systems analysis and definition;* (2) *systems*

design; (3) *implementation and unit testing;* (4) *system testing;* and (5) *operation and maintenance.*

Table 1.1. General models of software development

The waterfall approach	View a software process as being made up of a number of stages and the software process follows these stages one after another.
Exploring programming	Develop a working system as quickly as possible, and then modify that system until it performs in an adequate way.
Formal transformations	Develop a formal specification of the software system and transform the specification into a program.
End user computing	End users are responsible for development of their own systems.
Assembly from reusable components	Use existing reusable components to construct a new system. The development process is mostly assembly rather than creation.

There have been numerous refinements and variations of the original waterfall life cycle model. Figure 1.1 shows an evolution process of the waterfall model. The original model assumed that software development was a linear process. Later stages would not be started until the previous stages have been finished. However, in practice, the development stages overlap and feed information to each other. Each stage needs to go through several iterations before passing on to the next stage. Later stages may feed information back to the earlier stages and start a new iteration. This iterative process will continue until the software is phased out.

Software development often takes several months or even years to finish. In the original waterfall model, users will not see the system until after the implementation and testing stage. Because the cost of software modification goes up exponentially as the SDLC goes on, it will be too late and too expensive, if not impossible, for users to suggest modifications after the implementation. In addition, the original SDLC model assumes that a system is well specified prior to the system development. However, most users do not know what they want until they 'feel' and 'see' their system in operation

[Sprague 82]. The 'rapid prototyping' technique provides users with an operational version of system inputs and outputs. Once rapid prototyping is incorporated into the SDLC model, the users can 'feel', 'see', and play with the system prototype at early stages. Rapid prototyping often reveals omissions, inconsistencies and misunderstandings of system requirements. If the system does not meet their expectation, users may suggest modifications and see the changes quickly. Because system developers can get feedback from users much earlier than that of the old SDLC models, modifications become easier with less cost. Rapid prototyping also encourages users to become more involved and committed to their system from the earliest stages of development. Since users actively participate the development process, they are more ready to understand and accept the new system [Lantz 87].

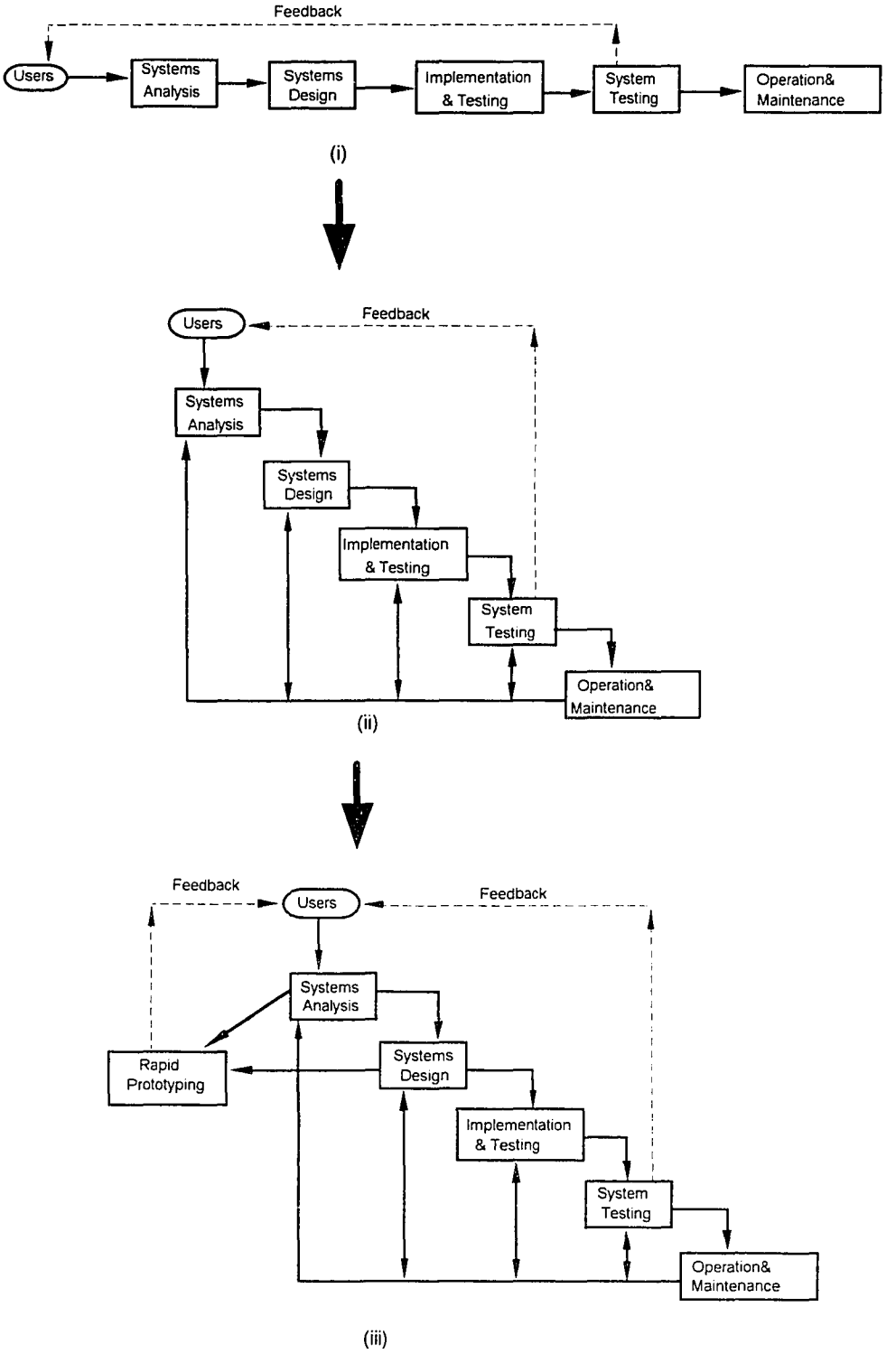


Figure 1.1. Evolution of the 'Waterfall' model

1.2 Tools for Structured Techniques

In the early days of information systems development, there were few tools other than the programming languages themselves. SE research has brought tools and design methodologies into every stage of the SDLC. The primary objective of these tools are: (1) to achieve high-quality programs of predictable behavior, (2) to create programs that are easily maintainable, (3) to simplify the programs and the programming process, (4) to speed up system development, and (5) to lower the cost of system development [Martin 88].

Based on these objectives, 'Structured Techniques' evolved from a coding methodology (structured programming) to techniques that include analysis, design, testing methodologies, and project management concepts. Martin and McClure [Martin 88] summarize the basic principles of structured philosophy as:

(1) *The principle of abstraction:* To solve a problem, separate the aspects that are tied to a particular reality in order to represent the problem in a simplified, general form.

(2) *The principle of formality:* Follow a rigorous, methodical approach to solve a problem.

(3) *The divide-and-conquer concept:* Solve a difficult problem by dividing the problem into a set of smaller, independent problems that are easier to understand and to solve.

(4) *The hierarchical ordering concept:* Organize the components of a solution into a tree-like hierarchical structure. Then the solution can be understood and constructed level by level, each new level adding more details.

Diagrams are often used in software development to illustrate the structures and ideas of analysis, design and implementation. Table 1.2 lists some structured diagram tools for different stages of the SDLC.

Table 1.2. A list of structured diagramming tools

Analysis Tools	Data Flow Diagram, Control Flow Diagram, Decision Table & Tree, Matrix, Dependency Diagram, Decomposition Diagram, HIPO Diagram
Design Tools	Structure Chart, Action Diagram, Wanier-Orr Diagram, Decision Table & Tree, Pseudo Code, Flowcharts, Screen layouts, Dialog Flow
Programming Tools	Flowcharts, Pseudo Code, Action Diagram, Decision Table & Tree
Database Tools	Entity Relation Diagram, Data Structure Diagram, Data Navigation Diagram, Logical Records, Physical Database and Files, Data Immediate Access Diagram, Data Dictionary

Among all the structured analysis tools, the Data Flow Diagram (DFD), developed by Demarco and Yourdon [Yourdon 79], is frequently used for function-oriented analysis to aid functional decomposition and process modeling. Figure 1.2 shows the four basic DFD symbols of Gane & Sarson method [Gane 79]. A 'Source or Destination of Data', also called 'External Node', is an entity outside the system that sends or receives data. A 'Data Flow' is a collection of data elements in motion. A 'Data Store' is a collection of data elements at rest. A 'Process' is an operation that transforms data. System models are created using these symbols in a hierarchical fashion starting with a single process context level diagram and 'exploding' the process to other more detailed levels. A Process explosion is the linkage of the process to a sub-DFD for purposes of functional decomposition and encapsulation. Processes in sub-DFDs can be further exploded until an atomic level is reached and the system has been fully, functionally decomposed.

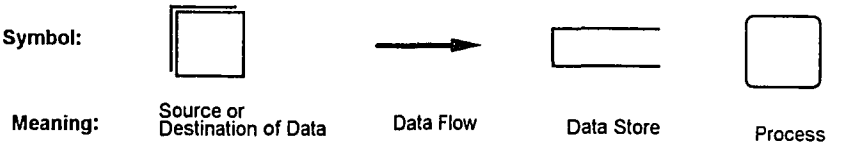


Figure 1.2. DFD symbols

Entity Relation Diagrams (ERDs) are often used for data-oriented analysis to describe the relationships of data entities and their attributes. Figure 1.3 is an example of an ERD using the Chen method [Chen 76]. An ERD has two basic symbols: an entity (the rectangle) and a relation (the diamond shape). Entities are collection of data describing an important element in the information system, such as 'Seller'. Relationships describe a logical connection between entities (e.g., owns). Cardinalities (numerical relationships between entities) are labeled on relationships. Each entity-relationship set is an expression of the underlying semantics of data. If properly constructed, an entity-relation set can be read like a sentence in either direction. For example, the relationship between 'Seller' and 'House' can be read as 'A seller owns M houses' and 'A house is owned by a seller'.

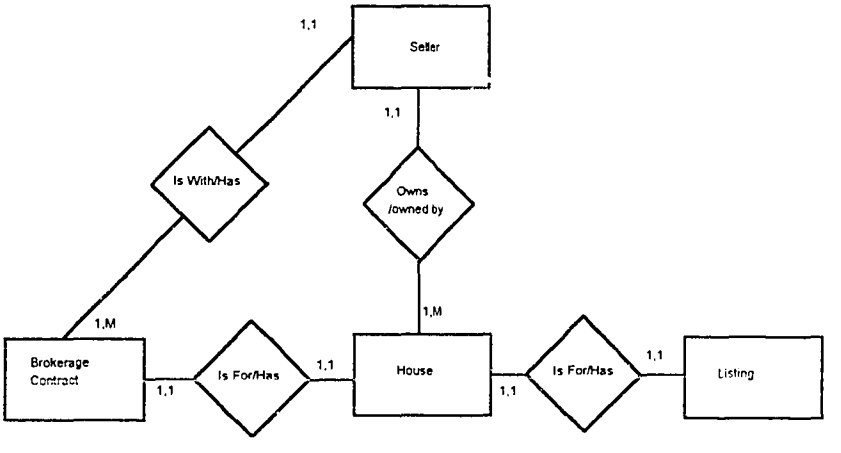


Figure 1.3. An ERD example

Function-oriented analysis and data-oriented analysis are complementary. ERDs can be used to describe data components in a DFD; and DFDs can be used to model data processing functions in data-oriented analysis.

1.3 CASE Tools

The objective of SE principles and methodologies is to improve the quality and productivity of software development. However, it is very hard to create, modify, and manage all kinds of diagrams and documents manually. Without computer-aided automation, the software engineering paradigms will not be practical in software development. McClure [McClure 89] concludes that: "Computer-aided software engineering (CASE) is the automation of software development."

Automation of the software development process is not a new concept. Since the early days of computer-based system development, assemblers and compilers have been used to translate high level language into machine language. CASE differs from previous computer-aided automation by enforcing software engineering methodologies in all stages of the SDLC. It encourages evolutionary and incremental development. Some SE concepts, such as abstraction, divide-and-conquer, hierarchical ordering, and step-wise refinement, are embedded in CASE. With CASE tools, software developers can derive the real benefit of SE methodologies.

There have been many CASE tools in the market for computer professionals. Most of these tools are based on structured techniques used to support structured analysis (SA), structured design (SD) and code generation. The recent development of object-oriented techniques brings a new perspective to the enforcement and implementation of basic SE philosophy. Several object-oriented CASE tools have been

proposed [Coad 90, 91]. It is believed that object-oriented techniques will be the dominant software development method of the next generation.

Although there are different variations, CASE tools have common goals to support software development [McClure 89]: (1) Provide an interactive development environment with rapid response time, dedicated resources, and early error checking. (2) Automate many software development and maintenance tasks. (3) Provide powerful user interface and visual programming capacity.

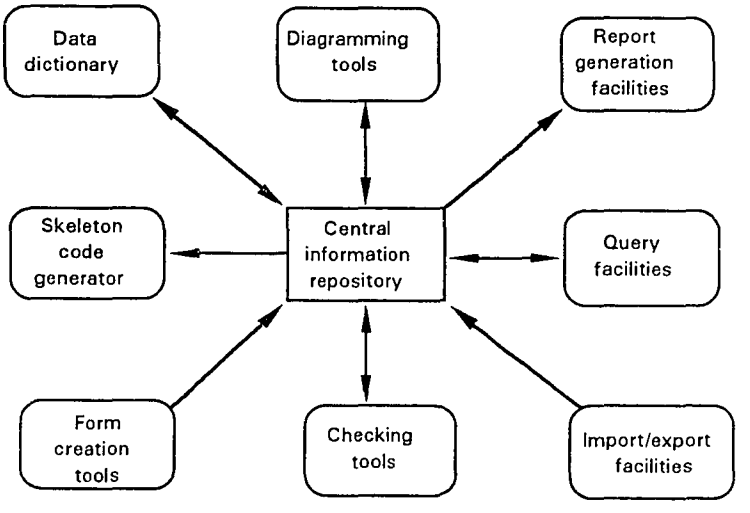


Figure 1.4. The structure of a CASE environment

Figure 1.4 shows the structure of a CASE environment. There is a central information repository as the information exchange center, from which other components store and retrieve information. A database is often used to facilitate the information repository. Other components can reside in the same computer as the repository, or distribute over a local area network.

A graphical diagramming capacity is fundamental to CASE to create DFD, ERD as well as other graphical models. A data dictionary is used to maintain names, labels, data attributes and other information of system models. Checking tools verify the validity

of the models. Query facilities and a report generator are used to browse information and examine completed designs. A code generator produces code or code segments from designs stored in the central repository. Form creation tools enable the users to customize the reports. Import-export facilities allow information interchange with other CASE tools.

Technologies utilized to build a CASE environment may change, but the basic functions that CASE provides to system developers should remain stable. Chen and Nunamaker [Chen 89] summarize the following basic functions of a CASE environment:

(1) *Elicitation:* CASE tools should be able to help system developers describe systems at analysis, design, and implementation levels.

(2) *Analysis:* CASE tools should be able to analyze the consistency and completeness of an elicited system description, detect errors or evaluate design alternatives.

(3) *Transformation:* CASE should help system developers to convert a system description from one level to another.

(4) *Information Storage:* Information elicited by system developers or generated by CASE must be stored so that project information can be shared and reused.

A complete CASE environment is a very complex piece of software. It may take years of team-work to finish. As a result, CASE tools are often very expensive and used mostly by computer professionals. Table 1.3 lists some information of commercial CASE tools available in the market [Oman 90]. These CASE tools support multiple SE techniques that help system developers in different stages of the SDLC. Unlike these fully functional CASE tools, this research focuses an architecture for front-end CASE

that only support the systems analysis stage of the SDLC. It will be less expensive, more compact and more user friendly to encourage user involvement in the development process.

Table 1.3. A list of some CASE tools as of 1990 [Oman 90]

CASE Tool	*Developer*	*Description*	*Platform*	*Price*
Cradle CASE	Yourdon Inc.	Supports all phases of the life cycle. Uses Yourdon structured method.	Sun , UNIX	$12,500 (single user)
Jackson CASE	Michael Jackson Systems	Maps closely to Jackson methods for system design	PC , DOS	$8.000
Card Tool	Ready Systems	An integrated set of requirements and design tool .	Sun , UNIX	$7,000
Excelerator	Index Technology	Provides multiple analysis and design tools to analyze, design and document IS.	PC , DOS Sun, UNIX	$8, 400
AdaGen	Mark V Systems	Supports both object-oriented and traditional Ada development.	PC, DOS Sun, UNIX	$7,850

1.4 Scope and Constraints of This Research

This section illustrates the need for more user involvement and dynamic model evaluation. This research tries to meet these needs by introducing hypertext and simulation into the system analysis process.

1.4.1 Making the user interface more friendly

The purpose of software engineering is to improve software quality. Like any other product, the goal of a software product is to satisfy consumer's needs. The definitions of quality have evolved from 'fitness of use' and 'conformance to established requirements ' to 'never ending improvement of product and service' and 'delight the customers'. From this perspective, a CASE tool should not only focus on how to

implement a design efficiently, but also the need to improve the user interface to communicate with users more effectively and earn their trust.

On the base of the SDLC coverage, CASE studies can be classified as 'upper CASE' (front-end CASE) and 'lower CASE' (back-end CASE) [Chen 89, 92a]. Upper CASE tools are used for information system planning, analysis, logical design, and some other user related aspects of system development. Lower CASE tools are used for physical design, code generation and program testing. Since user involvement and early error detection are critical to developing correct and cost-effective information systems, upper-CASE studies have drawn increasing attention.

Ives, Olson and Baroudi [Ives 83] have proved that greater user involvement leads to greater system success and less user involvement results in malfunction and unsatisfactory systems. To improve user involvement, many techniques have been proposed, such as Joint Application Design (JAD), Rapid Application Development (RAD), prototyping, and Group CASE [Yourdon 92].

Hypertext, which was originally designed for authoring and document management, is viewed as one of the enabling technologies for CASE [Chen 89]. Hypertext is a non-linear information network composed of information nodes, hyperlinks and navigation methods. Experimentations have shown that well-organized hypertext system may result in more friendly user interfaces and customized information retrieval [McAleese 89, Mynatt 92].

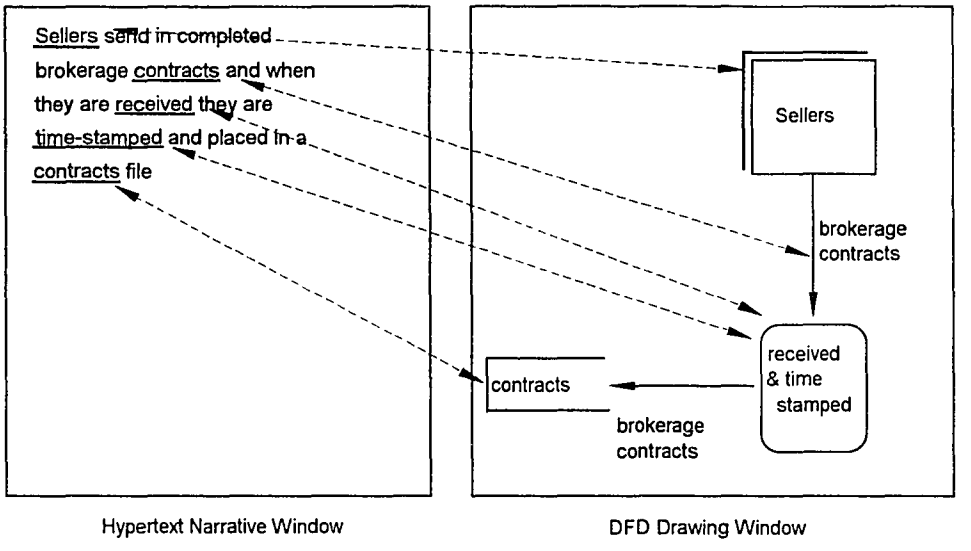


Figure 1.5. An example of hypertext linkages to a DFD

Figure 1.5. shows a structure to incorporate hypertext with a DFD. A text window holds the description narratives of a DFD model and another graphical window displays the graphical layout of the diagram. Hyperlinks are constructed while the model is been developed. The hyperlinks provide various ways to view the model. These links represent the thought path of the model designers (authors) and can be easily accessed by reviewers (readers). As a result, hyperlinks provide another dimension (hyper-dimension) in addition to the linear text and two-dimensional graphical model for communication between designer and users as well as among users themselves.

A user interface with multiple windows and hypertext conveys richer information to users. It is expected that once it is incorporated with a CASE tool, the hypertext interface will increase users' comprehension of system models. Such an improved channel of communication may help software developers to achieve more user involvement, less misunderstanding, and fewer errors in software development.

On the other hand, a WIMP (Window, Icon, Menu and Point) styled hypertext interface may help users not only to understand models designed by other people, but also to learn and to use the analysis techniques for their requirement specification. This will provide an opportunity to extend structured techniques and CASE, which are used mostly by computer professionals, toward end users. It will result in more user involvement, improved communication and better tools for user requirement specification.

1.4.2 Discovering system dynamics

In addition to the construction of system models, validation and evaluation of these models are also important CASE functions. Model validation is to check if a model is structurally valid and functionally correct. Model evaluation is to measure the performance of a valid model and weight different alternatives.

Most CASE tools check for mechanical errors in a model and study some static features, such as data balance, cardinality and data consistency. Noise and random factors have not been taken into consideration in static analysis. However, information systems are designed, implemented and used in a rapidly changing and turbulent environment. The activities of software development and application are dynamic by nature. Although prototyping techniques reveal some dynamic features and give a fast glimpse on the system's behavior, they can only uncover some of the *micro* features, such as what a user interface looks like, how it responds to users' requests and what functions may be included in the system. Some *macro* features which reveal the overall system performance, such as bottlenecks, job waiting time and resource utilization, still remain unknown. Analytical methods can be used to statistically analyze these performance features of system models [Ng 90]. Unfortunately, it is hard, if not impossible, to define mathematical formulas that precisely represent the behaviors of a system model. Studies

have shown that static analysis based on mathematical approximation may result in unrealistic conclusions in noisy and turbulent environments [Wild 91a]. Systematic dynamic modeling strategies are needed to counter noise and turbulence that may occur in system development [Sol 91].

Simulation is often used to model an unknown system and reveal the statistical features of the system performance when analytical solutions are not available. Simulation is non-destructive, repetitive and dynamic. It can be used to test system behaviors under different scenarios without implementation of the system. A study conducted by Warren [Warren 92] shows that simulation of IS models can provide better estimates of system performance and detect errors at early stages.

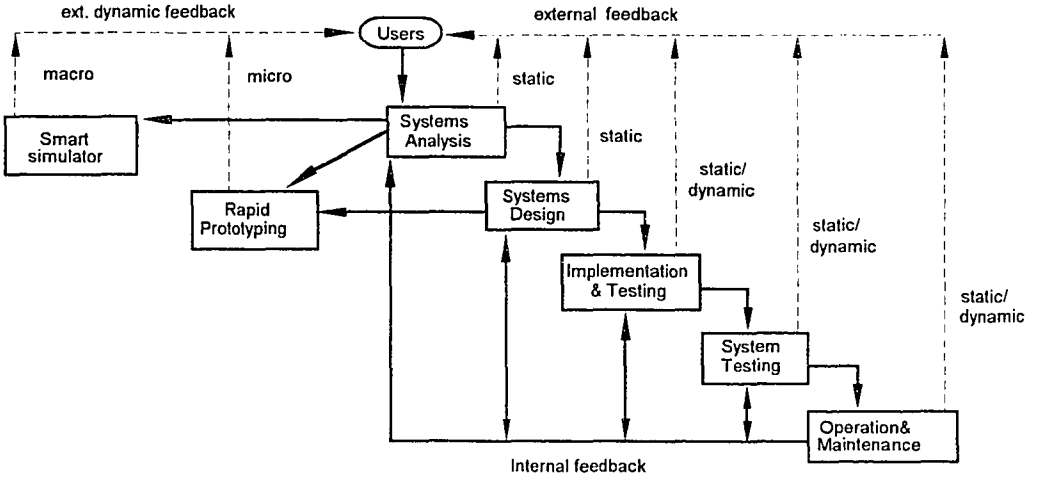


Figure 1.6. A revised SDLC model

Figure 1.6 shows a revised SDLC model with a simulation package embedded in the SE process. There are two kinds of feedback to users: *static feedback*, such as verbal or written reports and documents, and *dynamic feedback*, such as demos, prototypes and system previews. In the old SDLC model, users can, at most, get static feedback at the systems analysis and design stages. The system performance dynamics will not be

available until the coding and testing periods when it is too late and too expensive to make changes. The prototyping technique provides dynamic feedback on what the target system will look like to users at an early stage. Users may have the chance to offer suggestions for improvement when it is still feasible to change the system. However, prototyping only provide a *micro* view of a system. An embedded simulation package can be used to estimate the dynamics of overall system performance at an early stage and provide a *macro* view of the system at the systems analysis stage. The macro views of a target system are more important for strategic decision making, such as to determine the hardware and software configurations and to compare different design options. With information from dynamic evaluation, users and the system developers may have better control over the software development process.

To date, IS dynamic evaluation has not been routinely included in CASE tools. The potential of dynamic model evaluation may improve the IS development processes in the following ways:

(1) *Improve system performance estimation.* Unlike static evaluation, dynamic evaluation takes random factors into account and results in more precise estimations of system parameters under turbulent situations, such as job-load, service time, and average system response time.

2) *Improve hardware platform selection.* Because of improved system performance estimation, system analysts can reduce the chance of over-specifying or under-specifying hardware configurations. It may save hardware costs and avoid unsatisfied system requirements.

3) *Improve software environment selection.* Because system response time and data volume are better estimated, software systems can be chosen on a cost-effective basis.

4) *Improve re-engineering process.* Different re-engineering strategies can be compared through dynamic evaluation before any decision is made. It provides more control and quality assurance to the re-engineering team.

However, few users and system analysts are skillful at simulation techniques. They may neither have the time nor the experience to develop a simulation model every time an IS analysis problem occurs. They need help to determine simulation parameters and generate simulation models. Furthermore, simulation results are not always understandable and are very tedious. An expert system can provide suggestions and insights to help users and improve the effectiveness of simulation. O'Keefe described a different architecture incorporating a simulation system with an expert system [O'Keefe 86]. There have been other studies that incorporate an expert system in different simulation tasks [Fox 89, Hill 87, Park 90]. It is necessary and feasible to build an expert system that helps users to build simulation models and explain simulation results.

1.4.3 Levels of interactions

Software development involves different levels of interactions and activities. It ranges from highly abstract conceptual modeling to very detailed coding and testing. As described in the previous sections, the development is a team effort that can be viewed as a process full of interactions between 'authors' and 'readers', 'users' and 'analysts', and 'designers' and 'programmers'. There are different concerns at each level that require different tools.

Mittermeir and his colleagues add interaction into the water-fall model. They view software development as consisting of two dimensions: *interaction* and *refinement* [Mittermeir 90]. They propose 'outside in' and 'inside out' approaches to analyze the interactions of a system; and 'top down' and 'bottom up' approaches to refine the system functionality.

The interactions can be further decomposed into human to human interactions and human to machine interactions. In a study of hypertext and CASE integration, Oinas-Kukkonen focused on the human to human interaction among users and system developers as well as system developers themselves [Oinas-Kukkonen 93]. He concludes that hypertext is the right vehicle that serves as an intermediary among human actors.

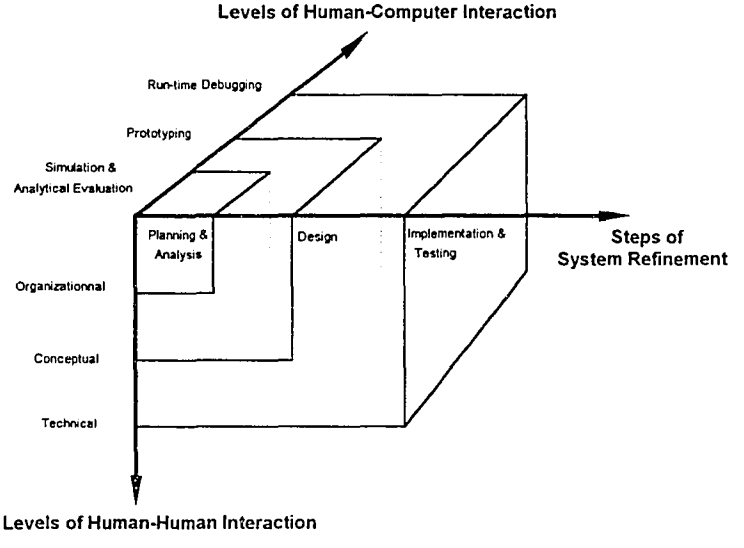


Figure 1.7. Different levels of interactions in system development

Figure 1.7 is a 3-dimensional model of interactions in software development. In the first two dimensions, human to human interactions are presented throughout the entire software development process. At the planning and systems analysis stage, human to human interactions focus on the organizational level. People are more interested in

which department is involved in the new information system, how the development team is organized, what is the budget for the project, what is the hardware and software configurations. At the later part of systems analysis and logical system design stages, information about the conceptual model of the proposed system needs to be circulated among the end users, the developers, the management, and other interested groups. At the implementation and testing stage, system developers are interested in technical details of how to implement the conceptual models, the test strategies that should be used, and the testing of code or functional modules. Hypertext is suitable for all the human to human interactions by providing hyperlinks among documents and improving understanding of the system.

In the human to computer interaction dimension, people must actively observe, study, and test the computer system and control it within their expectations. It is obvious that debugging and testing a real system is the most direct way to get feedback on system performance. However, at the systems analysis stage, the only tools available to study a non-existing system are simulation and some other analytical methods.

This project is to develop a tool that improves systems analysis by integrating hypertext to enhance human to human interactions, and simulation to enrich human to machine (models of a proposed information system) interactions. Since systems analysis is the corner-stone for later stages of the SDLC, improved system analysis greatly enhances system development as a whole.

1.4.4 Environments for software engineering and simulation

Software engineering and simulation are two distinct but closely related areas. Computer simulation involves the development of a simulation program. The principles and tools for software engineering can be used to direct the development of simulation

programs and simulation environments. On the other hand, as stated in the last section, simulation can be used as a tool to improve the software engineering process.

Each of the two areas has its own essentials. SE focuses efficient and effective implementation. Simulation is concerned with statistical analysis and model evaluation. Neither of the environments can replace the other. However, there are some commonalties between the environments of SE and simulation.

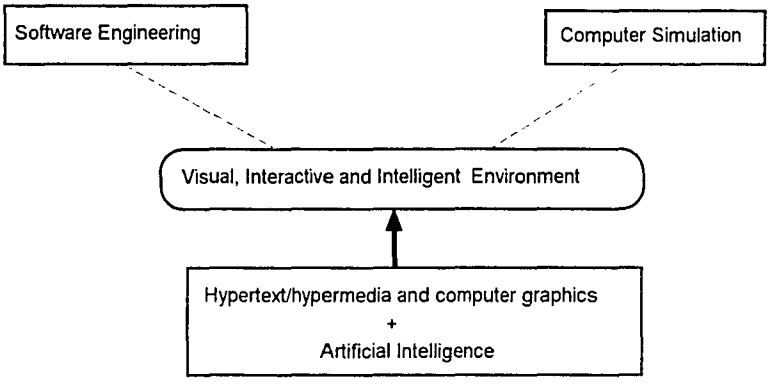


Figure 1.8. Software engineering and simulation

As hardware costs goes down and personal computing becomes more available, software engineering and simulation analysis require more distributed, collaborative and user friendly support environments. Figure 1.8 shows that enabling technologies, such as computer graphics, hypertext/hypermedia along with breakthroughs in hardware, become the foundation for a visual interactive environment. Artificial intelligence techniques provide vehicles for inference and intellectual reasoning of computer systems. There is a trend in software engineering and simulation toward the development of integrated, visual, interactive, and intelligent environments. The intention of this dissertation is to follow this trend and search for ways to create better systems environments.

1.4.5 What is HAT?

HAT (Hypertext Analysis Toolkit) is designed to address the problems found in upper CASE. A hypertext-based user interface is the fundamental component. It helps end users and system analysts to plan an information system, define the user requirements and analyze the system with DFD and ERD models. The goal of HAT is to encourage more user involvement, promote structured systems analysis methods, improve communications among users and system analysts, and estimate system performance at the early stages of the SDLC. This tool provides an opportunity to push traditional methods such as the use of DFDs and ERDs, which are practiced by computer professionals, toward end users and allow them to describe their needs in a more organized fashion. In addition, dynamic evaluation of information system models with an embedded simulation package may overcome some of the biases of static analysis and provide statistical estimation of the general system dynamics. Different alternatives can be judged and weighted before systems implementation.

Figure 1.9 shows an example of hyperlinks in HAT. Direct hyperlinks (dashed lines with arrow head) are constructed from requirement specifications of DFDs, ERDs and other documents. These links are connected as the analysis process is developed. Reviewers of the project can easily browse the models through hyperlinks and add new links.

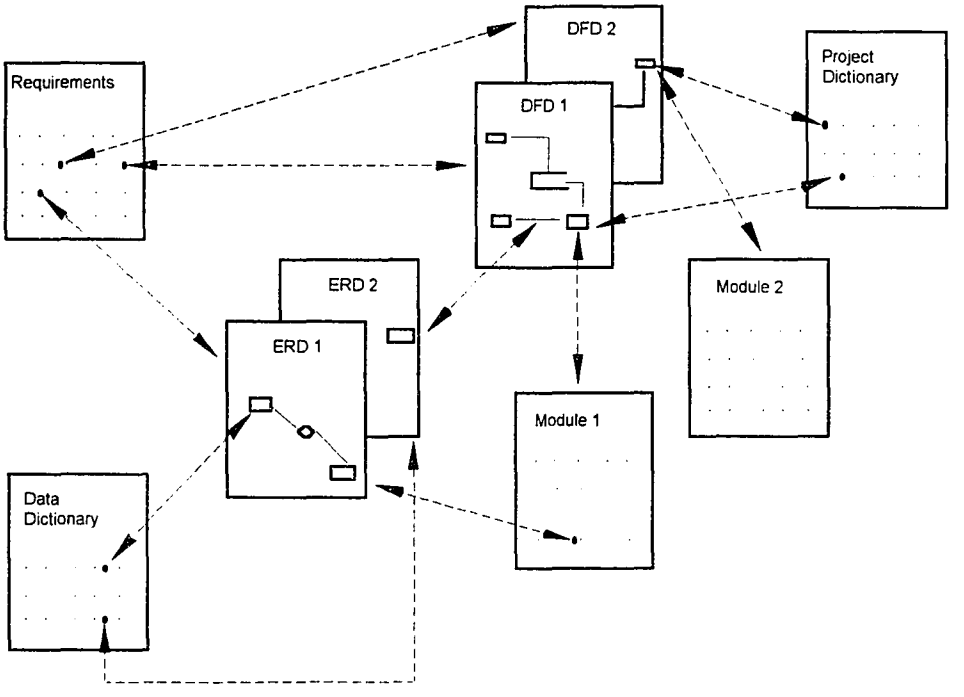


Figure 1.9. An example of hyperlinks among documents in HAT

The dynamic evaluation subsystem generates simulation models from DFDs and feeds the result back to the user interface. A simulation expert system functions as a helper to determine the parameters of a simulation model and explain the simulation result. Dynamic Data Exchange (DDE) links are used as communication channels among the user interface, the simulation package, and the simulation expert system.

The basic functions of HAT can be listed as:

(1) A WIMP (Window, Icon, Menu and Pointer) styled GUI (Graphical User Interface) for friendly, easy user access, which includes:

(a) A graphical editor that provides graphical tools for drawing and editing DFD and ERD models.

(b) A hypertext editor that provides tools to create and delete hyperwords and connect them with graphical objects in DFD and ERD models.

(c) A browser and navigation methods for model query and retrieval.

(d) Interfaces to access the data dictionary and data model descriptions.

(e) A report generator that generates reports from DFD and ERD models in a pre-defined format.

(f) Interfaces for simulation modeling and parameter setting.

(2) Model validation and evaluation subsystem:

(a) A static evaluator of DFD models that checks for mechanical errors in a DFD drawing and data flow balance.

(b) A dynamic evaluator of DFD models that generates simulation models from DFD model scripts and runs a simulation to get statistics of system performance.

(c) A simulation supporting expert system that aids in simulation model development and explains simulation results.

This research is an exploration of ways to integrate hypertext and simulation with a CASE environment. These are two enabling techniques that may improve CASE performance and software productivity in the future. HAT focuses exclusively on the systems analysis stage and is an aid to effective communication with structured systems development methods. HAT itself cannot work as a stand-alone CASE tool. However, it may serve as a front-end CASE tool and a tutorial tool for educational purposes.

CHAPTER 2 LITERATURE REVIEW

The objective of this literature review is to support the underlying concepts of Hyper Analysis Toolkit (HAT) and lay the foundation for its creation. This review includes an explication of (1) the concepts of upper CASE and user-oriented techniques; (2) an application of hypertext to CASE usage; (3) simulation systems and embedded simulation in CASE; and (4) an integrated simulation environment with visual agents and expert systems.

2.1 Upper CASE Studies

Software development is a labor-intensive process that ranges from several months to many years. With the advance of hardware technology, especially microcomputer technology, the cost of hardware is decreasing and CPU time is no longer a scarce resource. Thus, sophisticated user interface technologies become practical. In addition, new developments in software methodologies make it possible for CASE and other software tools to support functions such as 4GL (4th Generation Language), automatic code generation and automatic testing. As a result, software developers are greatly relieved from the burden of programming and code optimization. However, this does not mean the end of software crisis. The demand for software has grown much faster than the improvement in the productivity of software engineers. While new software development techniques relieve some programming and testing urgency, overall time pressure for complex systems project completion is increasing. More efforts are required to not only increase the efficiency of programming, but also to improve the effectiveness of software development, especially in the early stages of the SDLC.

Martin [Martin 88] identifies the principal source of software bugs as being in the stages of analysis and design, not in the coding process itself. As shown in Figure 2.1, 60% of software bugs come from systems analysis and 85% from analysis and design combined. The study also showed that 80% of the time and effort required to locate and debug software problems were due to logic errors. It is obvious that software development time and cost will be significantly reduced if errors in analysis and design are detected and corrected soon after they occur.

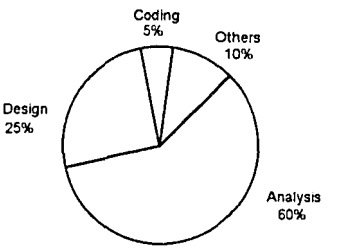


Figure 2.1. Sources of software bugs as presented in [Towner 89]

Since the quality of systems analysis and design is of great importance, much CASE research has paid special attention to upstream activities, such as analysis, logical design, and some other user related aspects [Chen 89, 92a]. Chen further states that because of the increasing demand for business application and widespread use of microcomputers, the traditional SDLC model has been expanded to include business and information system planning activities. More users and field experts are involved in the planning and preliminary analysis stages. In addition to structured analysis and design tools, it requires upper CASE to provide IS planning tools representing high level business objectives and organization structures, as well as their relationships with business functions. These planning tools should allow users to navigate through descriptions (in graphs or forms) of a system in various related aspects via hypertext-styled links. Consistency within and across modeling aspects should be checked and ensured by upper CASE.

Table 2.1. Summary of user involvement from [Ives 84]

Features	***Descriptions***
Type of participation	1. *Consultative:* Design decisions are made by system group. But the objective and the form of the system is influenced by the needs of the user department. (indirect involvement) 2. *Representative:* All levels and functions of the affected user group are represented in the system design team. (direct involvement) 3. *Consensus:* An attempt is made to involve all workers in the user department, at least through communications and consultation, through the system design process. (highly direct involvement)
Degree of participation (the amount of user influences over the final product)	1. *No involvement:* Users are unwilling or not invited to participate. 2. *Symbolic involvement:* User input is respected but ignored. 3. *Involvement by advice:* User advice is solicited through interview or questionnaires. 4. *Involvement by weak control:* Users have 'sign off' responsibility at each stage of system development process. 5. *Involvement by doing:* A user as design team member or as liaison joins the information development group. 6. *Involvement by strong control:* Users directly pay for the new development out of their own budgets.
Outcome of user involvement	1. *System quality:* More user involvement results in improved understanding of the system, improved assessment of the system needs, and improved evaluation of system features. 2. *System acceptance:* More user involvement increases user perceived ownership of the system, decreases resistance of change, and increases commitment to the new system.

Ives and Olson define the term 'user involvement' as '*the participation in the system development process by representatives of the target user group*' [Ives 84]. They indicate that the concept of user involvement can be traced to the theories and research in Organization Behavior, including group problem solving, interpersonal communication and individual motivation. Table 2.1 gives a summary of user involvement research.

Gould and Lewis [Gould 85] have also observed the complexity of IS development and the importance of user involvement. They assert that nobody can get it right the first time and IS development is full of surprises. Furthermore, developing a user-oriented system requires living in a 'sea of changes'. Ignoring the changes does not eliminate the need for change. They recommend three principles of design: (1) Early focus on users and tasks – understand user needs at the very beginning, (2) Encourage the intended users to use prototypes and a simulation of the system, (3) Empirical measurement – observe, record, and analysis of user response, and (4) Iterative design – go through the cycle of design, user test, measure, and redesign.

In practice, the needs for more interactive tools and more user involvement have been widely recognized. Users are becoming involved at discrete points in the SDLC in various forms, namely review, sign-off meetings, and weekly steering committee meetings. These common techniques, however, are still insufficient to create enough user involvement [Gould 85]. There are several reasons that more powerful, structural and informative communication channels need to be created— (1) system analysts often do not have the experience nor the expertise to fully understand the business, the people and the politics in an application field, (2) as systems sophistication progresses, there has been an increasing focus on the introduction of creative and innovative ideas into information systems (often users, not designers, are innovators who bring novel solutions to their own problems), (3) user requirements may change over time during the lengthy

software life cycle. There should be enough leverage for system developers to be well informed of changes and reflect the changes in their design in a timely fashion.

Joint Application Design (JAD) presents some solutions to the problems. JAD was originated at IBM in the late 70's. By the mid 80's, JAD gained recognition in the IS community as an effective method to manage analysis and design stages in the systems life cycle [August 91]. More recently, Martin [Martin 90a] created a variation of JAD known as Rapid Application Development (RAD). IBM has built a very large program to support JAD, both for its customers and for its massive internal development needs. The principles of JAD are to introduce structures and formats for 'how to run a design meeting'. Once the principles were supported by software and built into the SDLC, JAD became a methodology that centered on other activities from IS planning to system maintenance. Although there has not been a rigid set of rules nor a single structure, JAD provides 'soft' guidelines to solve some user involvement problems. Carmel, George and Nunamaker [Carmel 92] summarize some JAD solutions as shown in Table 2.2. At the systems analysis stage, the JAD solution is to encourage teamwork among the users to define system requirements and some design details.

Table 2.2. JAD solutions to user involvement problems as presented in [Carmel 92]

User involvement problems	***JAD solutions***
System analysis	Have the users define the requirements and some design details, synergy of group work.
System innovation	Encourage creativity, brainstorming, pool experts together
Fluctuation of requirements	Gather all decision makers in one place, group dynamics

There have been several recent studies in the incorporation of JAD with CASE, group support systems (GSS), and electronic meeting systems (EMS) to achieve better user involvement [Carmel 92, Liou 93]. The research takes advantage of state-of-the-art

computer network and group software techniques to complement the lack of user attention found in some CASE tools and make JAD methods more accessible to users. The Electronic JAD (E-JAD) proposed by Carmel provides GSS 'tool boxes' to support traditional JAD sessions. Each 'tool box' is carefully orchestrated by a facilitator who plays an active role in meeting planning and in supervising the meeting process. Users can join JAD sessions from distributed locations and use tools in a given 'tool box'.

2.2 Hypertext and CASE

2.2.1 Hypertext and hypermedia

The term 'hypertext' was first coined by Ted Nelson in 1965. Nelson attributes the underlying concept to Vannevar Bush and his Memex system in 1945. An easy explanation of hypertext might be 'a non-linear network of linked information nodes'.

> Hypertext is non-sequential reading and writing that allow authors to link information to create paths of related materials, annotations and existing text.
>
> -- Jeff Conklin [Conklin 87]

Hypermedia is an extension of hypertext where the information nodes may contain not only text and graphics but also sound, video and animation. There is not a clear-cut distinction between hypertext and hypermedia. Hypertext is a more general term. Discussions of hypertext are also suitable for hypermedia.

Conklin [Conklin 87] discusses three different views of Hypertext: *View of Linked-ness, View of Nodeness* and *View of Navigation.* These different views represent features in a hypertext system.

- *View of Linked-ness:* A Hypertext system is an information network connected by hyperlinks. These hyperlinks encourage writers to make references and readers to make their own decisions. With hyperlinks, a hypertext system has the

feature of nonlinear information retrieval that users can start from anywhere in the information network and choose whatever ways they want to search for information. Because of computer support, hyperlinks can be determined at run time and it is very fast to go along hyperlinks to visit information nodes in the network. There can be different kinds of hyperlinks in a hypertext system which serve different purposes. They can be: *Referential Links, Organizational Links, Conditional Links,* and *Activation Links.*

- *View of Nodeness:* Information nodes are the information carriers of a hypertext system. An information node has natural correspondence with an object in the real world. For example, we can view 'computer' as an information node and 'peripheral' as another information node.

- *View of Navigation:* A set of navigation methods that guide users going through the information network in a hypertext system. Users have the liberty to decide where to go and how to go. The navigation method that a user applies may have a direct impact on the result of information retrieval. Table 2.3 is a list of navigation strategies and their usage.

Nelson [Nelson 87] points out that human thinking is not sequential but is based on associations. It is often not reasonable to make every reader of hypertext read all the materials in the information network. A hypertext system should allow its reader to choose according to his/her interest.

Table 2.3. User preferences of navigation methods as presented in [McAleece 89]

Navigation methods	*Text*	*Graphics*
Scanning	**	**
Browsing	***	*
Searching	**	*
Exploring	*	**
Wandering	*	***
Preferences: *: small, **: some, ***: high		

Scanning covers a large area but without depth.
Browsing follows a path until a goal is reached.
Searching has an explicit goal and strives to find it.
Exploring finds out the content of a given information web without a pre-defined goal.
Wandering goes through the information web purposelessly and unstructurally.

As a non-linear information storage and retrieval method, hypertext has great potential to be used in many areas from authoring and document management to tutorial and entertainment. New developments in hardware, such as CD-ROM, high resolution color monitors, fast microcomputers, allow for the creation of hypertext and hypermedia applications. Some common features of hypertext and hypermedia applications can be listed as following [Conklin 87]:

- *Easy to trace reference:* Hyperlinks can be constructed among information nodes. Non-linear tracing along the hyperlinks can easily go from one node to another.

- *Easy to create new references:* Most hypertext authoring systems allow users to add and delete hyperlinks.

- *Information structuring:* Information nodes are connected by hyperlinks in hyper-dimension space. They can be structured in many ways: tree, graph, hyper-cube, hyper-tree and so on. These structures break the limit of linear printed text and 2-D graphics so that they can store and retrieve information more effectively.

- *Global views:* Browsers, indexes and maps are common navigation tools provided by hypertext systems. Users can have global views of the whole document as well as their own positions in the hypertext web.

- *Customized documents:* Users have the maximum liberty to go through the information web with navigation methods provided by the system to any reachable node. They may create their own search patterns and information sub-net in the web.

- *Modularity of information:* Each information node can be viewed as an autonomous unit, like an object. It will not be affected by changes in other nodes.

- *Consistency of information:* A well-organized information network should have no redundant nodes. All references go through hyperlinks.

- *Information collaboration:* Hypertext can be a shared information resource on which multiple users can work synchronously or asynchronously.

- *Two problems - disorientation and cognitive overhead:* Because hypertext and hypermedia systems contain so many nodes and their structures have so many variations, a user may 'get lost' in the jungles of information. To solve this problem, browsers, indexes and maps are often used to guide users. These tools may also help users to ease some of the cognitive problems so that users don't have to always remember the paths they have gone through.

How effective is hypertext as an information retrieval tool? Some other studies have proven that users read electronic documents more slowly than they read the same documents on paper [Wright 83]. Mynatt and his colleagues [Mynatt 92] compare the recall and readability of hypertext and printed book. A hypertext encyclopedia and an

identical printed copy of the encyclopedia are used for the experiments. The result shows that hypertext was not superior over printed document for linear information retrieval. However, when non-linear retrieval and more complex questions are involved, the performance of hypertext users is better than that of printed documents users.

2.2.2 Apply hypertext to IS development

Software development can be viewed as a special authoring process. The products from this process are computer programs and software documents. The management of these documents is one source of software development problems. It is quite natural to employ powerful document management tools like hypertext to improve software document management. Chen [Chen 89] lists hypertext as one of the enabling techniques for the next generation of CASE tools.

Much of the difficulty in developing and maintaining a large software system is inherent in the complexity of the tasks themselves. Brook [Brook 87] points out that the complexity is the essential property of software and part of the problem is the inability to visualize the system . Although Brook asserts that software structures cannot be visualized because they contain higher dimensions of space, Carando argues that a lot of the work in software design and analysis goes into conceptualizing these elements in visual forms [Carando 89]. She points out that these limited attempts at visualization, if not completely correct, are at least of some help in finding a direction to explore. Furthermore, hypertext alleviates some of the problems by providing (1) a repository for all project information, (2) hyperlinks that describe internal relationships, and (3) an interface that improves software visualization.

Hypertext tries to imitate associative human thinking and offers a flexible solution for managing various kinds of media (text, graphics, audio and video). Hypertext can empower a CASE environment in the following ways [Oinas-Kukkonen 93]:

(a) *Graphical user interface*. Users may visually and interactively select and/or define hyperlinks and zoom the information 'lens' to the desired level of details. Different browsers of the hypertext system provide orientation of global views that guide users through the information network.

(b) *Supporting data/document repository*. Hypertext can provide efficient data and document storage and retrieval to integrate the file management in a CASE environment. An advance hypertext system also supports document generation and version controls that can further improve CASE performance.

(c) *Representing semi-structured information*. Hypertext combines the semantics of natural language and a node-link structure. It separates the logical and physical structure of a document and makes it easier to capture the reasoning process behind a design decision.

(d) *Support collaboration*. A Hypertext database provides a common *hyperspace* that can be shared by different parties.

The history of hypertext applications in IS development extends over 20 years ago when the *Augment* system was developed [Conklin 87]. Several examples of hypertext systems that specially target applications for software engineering are as follows:

Augment, developed at Stanford Research Institute, was the first hypertext system applied to SE. Its research objectives were to develop principles and techniques

for designing an 'augmentation system' that conceptualize, visualize and organize working materials. Incarnation of these ideas in current software products provides office automation support for SE, including document preparation and the journalizing of electronic mail entries [Carando 89, Conklin 87].

Neptune of Tektronix has an open layered architecture. *Neptune* has two distinct parts: the front end – a SmallTalk user interface, and the back end – a transaction-based server called the Hypertext Abstract Machine (HAM). HAM is a generic hypertext model which provides operations for creating, modifying, and accessing nodes and links. It maintains a complete version history of each node in hyper-documents and provides rapid access to any version of a hyper-document. It also provides multi-user access over a computer network. The Dynamic Design function in HAM allows great flexibility for users to construct nodes and links [Bigelow 88, Conklin 87].

Shadow, developed at Schlumberger Laboratory for Computer Science, incorporated AI with hypertext and SE. It represents software projects as a series of models. Users can define annotations and relations among models and model components. *Shadow* provides a highly visual, directly manipulatable interface that supports element linking and traversal of the model network. In addition, *Shadow* has a knowledge acquisition scheme that automatically captures information augmenting elements, models and the links between them [Carando 89].

ISHYS and ***DIF***, a ten-year-old project at the University of Southern California, supports hypertext document management in the SDLC. *DIF* (Document Integration Facility) provides all the basic functions of hypertext retrieval and software version management. In addition, it also provides consistency and completeness checking of documents, on-line software document inspection, as well as intra- and inter-document

tracing. *ISHYS* (Intelligent Software HYpertext System) adds intelligent behaviors to *DIF*. It contains knowledge of the surrounding environment and has the ability to: (1) automatically determine attributes of hypertext nodes, and (2) coordinate and schedule agent tasks in the software life cycle [Garg 89].

HyperCASE, an on going project of Amdahl Australian Intelligent Tools Program, integrates a collection of tools to provide a visual integrated and customized SE environment. It consists of loosely coupled tools for both text and diagram presentation. *HyperCASE* combines a hypertext-based user interface with a knowledge-based document repository. It also includes an extensive natural language capacity tailored for the CASE domain [Cybulski 92].

Hypertext Intermediary Agent, is an on-going hypertext-CASE integration project conducted by the Department of Information Science of University of Oulu, Finland. In the preliminary report of this project, Oinas-Kukkonen [Oinas-Kukkonen 93] points out that a CASE environment itself is an evolutionary information system during the development of the target IS. The communication and data gathering function during this evolution becomes more and more problematic among different actors and actor groups. He describes the use of a collaborative hypertext system as an intermediary agent that conveys information among different parties. Figure 2.2 shows the basic idea of a hypertext intermediary agent. There are four actor groups in this scenario: CASE developers, CASE users, IS developers, and IS users. The people participating in each process can use (reading) and alter (authoring) hypertext according to their needs. The collaborative hypertext system allows all participating groups to share a common *hyperspace*.

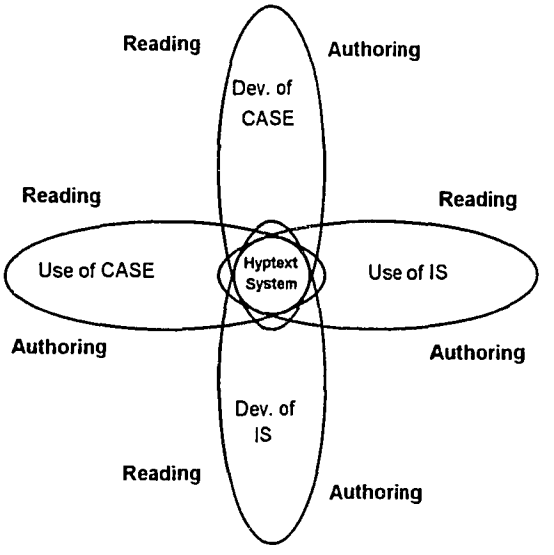


Figure 2.2. Hypertext as an intermediary agent in IS development

In this hypertext intermediary agent paradigm, an embedded hypertext system improves communication and document management at three different levels: (1) the *organizational level*, which defines the organizational role and context of the target IS; (2) the *conceptual level*, which defines an 'implementation-independent' specification of the IS; (3) the *technical level*, which defines the technical implementation of the IS. As a result, the hypertext system becomes the center of communicative software in a CASE environment and serves as an intermediary agent between other software agents and different human actors.

The current trends in hypertext applications in IS development can be summarized as:

(1) Encourage user-oriented authoring and offer great flexibility for users to manipulate documents. With a hypertext system, users have the freedom to choose what to read and how to read. They can also alter the document for their own need.

(2) Emphasis on highly interactive user interface and the visual expression of software models. Hypertext is ideally suited for designing and implementing adaptive user interface.

(3) The use of richer media. Earlier systems, like *Augment* and *Neptune*, are basically text-based system with different browsers. Later systems like *HyperCASE*, include not only text but also diagramming tools.

(4) The use of AI techniques to automatically generate hyperlinks, identify attributes and support the modeling process.

(5) The coverage of more than one stage of the SDLC and whole-life-cycle document management. A hypertext system can be the information exchange center that conveys information for different people at different levels.

(6) The inclusion of consistency checks and validation capabilities.

Table 2.4. Comparative listing of hypertext and CASE as presented in [Cybulski 92]

Hypertext	*CASE*
Document authoring	Diagram editing, Text-oriented tools
Browsing and navigation	Traversing through program modules and refinement levels
Document aggregation	Module libraries, Data structure groups
Virtual structures	Code generation
Dynamic computation	Run time results
Revision management	Software configuration management
Group work	Project team development
Extendibility / tailor-ability	Multiple methodologies
Concept annotation	Design decision recording
Consistency checking	Validation / verification
Completeness assessment	Project plan tracking

These examples above show that hypertext applications in the SE process are feasible and beneficial to software development. Table 2.4 compares functions of hypertext and CASE. It is clear that hypertext can be applied to CASE in diverse ways to solve or alleviate systems development problems.

2.3 Simulation and IS Dynamics

2.3.1 Dynamics of information systems

An information system is a cohesive part of an organization. An information system is composed of people, hardware, software, data, and procedures. It collects, transmits, processes, and stores data. It also retrieves and distributes information to various users in an organization [Ahituv 90].

Because an information system is not isolated, it is subject to change with its environment. Our society is dynamic by nature and organizations in the society must accommodate these dynamics. Unfortunately, current information systems development is strongly linked to the uses of static models, which often fail to express the system dynamics. Many IS problems can be traced to the nature of IS design and its impact on our way of thinking and modeling [Sol 91]. In addition, an information system itself is also dynamic by nature. A software system has different functions to perform different missions from different initial points and variable input data [Cobb 90]. New perspectives and dynamic modeling methods are needed to cope with the dynamic aspects of information systems.

IS dynamic modeling depends on two major factors [Warren 91]:

(1) *The dynamic capability of system components:* the rate at which jobs can be processed. System components include combinations of computer systems (databases, operating systems, application software, and computer hardware), non-computer factors in the operation of the IS, and people.

(2) *The design of the IS:* (a) how does work flow through the system components? and (b) what are the performance requirements and objectives of the IS, such as response time, available resources and budget limit?

The goal of IS dynamic modeling is to determine the extent to which a set of system components in an IS design satisfies the performance requirements of that IS design.

2.3.2 Apply simulation to discover IS dynamics

Queuing networks are frequently used to model complex systems, such as production systems, communication systems, computer systems and flexible

manufacturing systems. To describe the behavior of a queuing system, five basic characteristics of the process need to be specified: (1) the arrival pattern, (2) the number of servers, (3) the service pattern, (4) the server discipline, and (5) the system capacity. A queuing network model can represent time and stochastic factors of a system, which are fundamental to dynamic modeling.

Queuing network problems can be solved by mathematical formulas or discrete-event simulation. For many complex systems, there is no feasible analytical solution derived from mathematical formulas. Simulation of a mathematical model becomes the only way to solve the problems [Law 91]. Unlike the testing of physical models, analysis of mathematical models is non-destructive and repetitive. This distinction, when applied to IS performance evaluation, can be translated to: (a) *physical model experiment*: testing with a real computer system, and (b) *mathematical model experiment:* simulating a queuing model. Simulation provides a way to test relevant aspects of a real system or hypothetical system without actually building the system. This feature is ideal to evaluate IS designs at an early stage when the actual system has not been developed.

Developing a simulation model is not trivial. It requires the same amount of effort, attention and discipline as developing any other computer application. An invalid or inaccurate simulation model may lead to wrong conclusions. Figure 2.3 describes ten basic steps to carry out a simulation study [Law 91, Widman 89]:

(1) *Formulate problem and plan the study:* Every study must begin with a clear statement of the study's overall objective and specific issues to be addressed; without such a statement there is little hope for success.

(2) *Collect data and define a model:* Information and data should be collected on the system of interest (if it exists) and used to specify operating procedures and probability distributions for random variables used in the model.

(3) *Logic model validation:* When the model has been specified, it must be checked with the decision maker and the intended model user to make sure that the model represents the real system accurately and completely within the application domain.

(4) *Construct a computer program and verify:* Select a simulation language or general purpose programming language to construct a computer program based on the model and verify that the program correctly represents the model's function.

(5) *Make pilot runs to verify the computer program:* Pilot runs of the verified model are made for validation purpose.

(6) *Extensive model validation:* Pilot runs can be used to test the sensitivity output to small changes in an input parameter and improve the model if necessary.

(7) *Design experiments:* It must be decided which system design to simulate if there are some alternatives. Decisions have to be made on such issues as initial conditions for simulation runs, the length of warm-up period, the length of simulation runs and the number of independent simulation runs. Different scenarios have to be prepared on how the input parameters are going to change.

(8) *Make production runs:* Production runs are made to provide performance data on the system of interest.

(9) *Analyze output data:* Statistical techniques are used to analyze the output data from production runs. Typical goals are to determine a confidence interval for a measure of performance or to decide which simulated system is best to some specific measure of performance.

(10) *Document, present, and implement result:* It is important to document the assumptions of a simulation model. The result and recommendations should be presented to users.

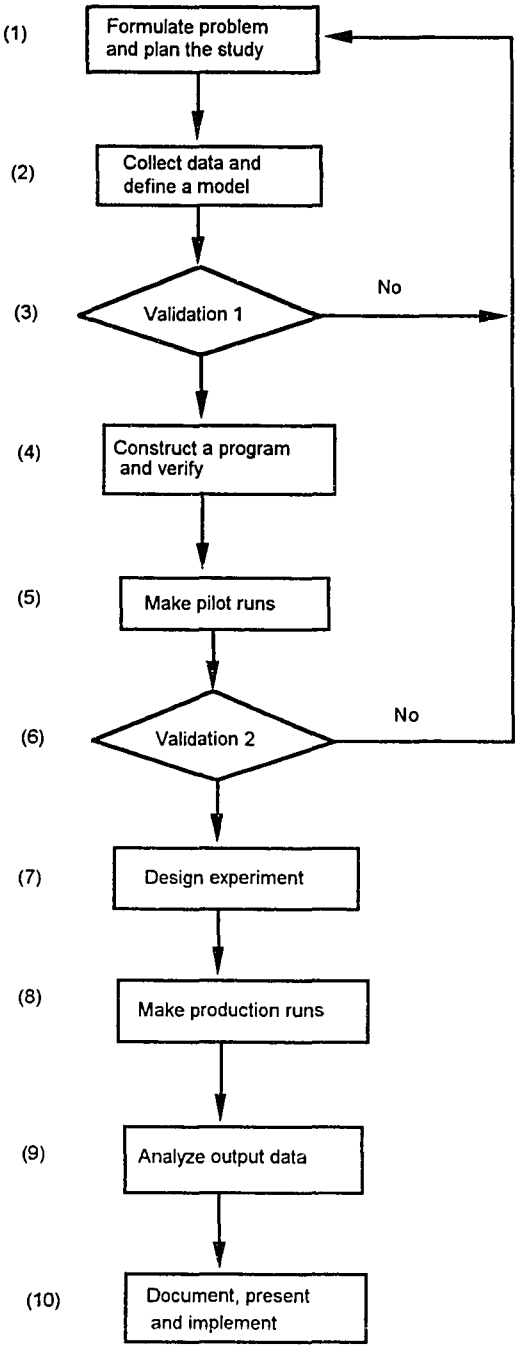


Figure 2.3. Steps in a simulation study as presented in [Law 91]

The relevance of a queuing network model and IS dynamic evaluation can be justified as: (a) an IS will entail a number of processes where often the input of one process is the output of one or more other processes, (b) the rate at which a job is completed in one process and sent to the next process can be estimated, (c) the amount of time that jobs spend waiting at some processes is an interesting parameter for evaluation. The dynamic aspects of an information system can be thought in terms of a network of queuing systems where the outputs of some of the queuing systems connect to the inputs of some of the others [Warren 91].

System simulation has been broadly applied in business. In fact, it has been the second most frequently used Operations Research (OR) technique (after statistical analysis) [Law 91]. The following sections review several successful simulation projects for information systems.

Eich [Eich 89] presents a methodology for the simulation of database architecture for performance evaluation. It is implemented in SLAM II and can be modified readily to accommodate different architectures. It has three major components: (a) a basic simulation model which defines the basic environment but does not detail the hardware / software components, (b) a system simulation model which defines the details of hardware / software configurations, (c) an execution model which describes the exact parameters to be examined and the simulation experiment design. Users can specify the system simulation model at any desired level of detail.

QASE of Advance System Technologies allows users to graphically depict software running on a hardware environment so that developers can study how certain applications affect a proposed system's performance [Gore 90]. QASE allows modeling

of wide area networks, where the type of hardware and operating system of each computer can be specified. Performance evaluation can be derived rapidly by simulation.

Eddins [Eddins 90] proposes a method to enhance the traditional DFD approach with some dynamic features and incorporate simulation with the traditional structured analysis and design process. An expanded DFD model is converted into an equivalent simulation model in SIMSCRIPT. Evaluation of the system dynamics can be achieved from the simulation results. The research suggests that CASE should include the capacities of simulation modeling and simulation analysis.

Wild and Griggs [Wild 91a] conducted research to compare the difference between static and dynamic analysis of DFD models in a noisy and turbulent environment. A SIMAN simulation model is derived from a DFD. The result shows that in a turbulent environment, static analysis may yield unrealistic results while simulation analysis can still capture the dynamic and probabilistic features of the system.

Warren has developed a prototype of a CASE/Simulation system [Warren 92]. The system imports a DFD model from a CASE tool and converts the DFD model automatically into a simulation model for dynamic evaluation. The prototype runs under X Windows on a Sun Workstation. It automatically interprets a DFD model as a queuing network and conducts simulation under specified parameters. A knowledge-based help support system is built into the prototype to provide a model-based expert advice in simulation modeling and simulation output interpretation. A series of behavioral studies of system developers based on this prototype have shown that dynamic evaluation of DFD models lead to more accurate assessments of IS design dynamics.

2.3.3 AI and simulation

The incorporation of AI into simulation is twofold – first, the desire to make simulation methods easier to use and more widely available, and second, the need to model increasingly complex systems, particularly systems that include some elements of human decision making [O'Keefe 89]. Widman [Widman 89] asserts that building and using a simulation model is a skilled process requiring expertise in a number of theoretical fields including statistics, system analysis, and numerical analysis. In addition, experience is needed to use simulation as an effective tool. For these reasons, an expert system can be applied to the simulation process. Furthermore, O'Keefe shows that AI workers have an increasing need to include simulation in AI systems so that the effects of a decision can be extrapolated over time and an expert system can use a model of a system to aid in reasoning. He concludes that the interdisciplinary application of AI and simulation is natural and practical.

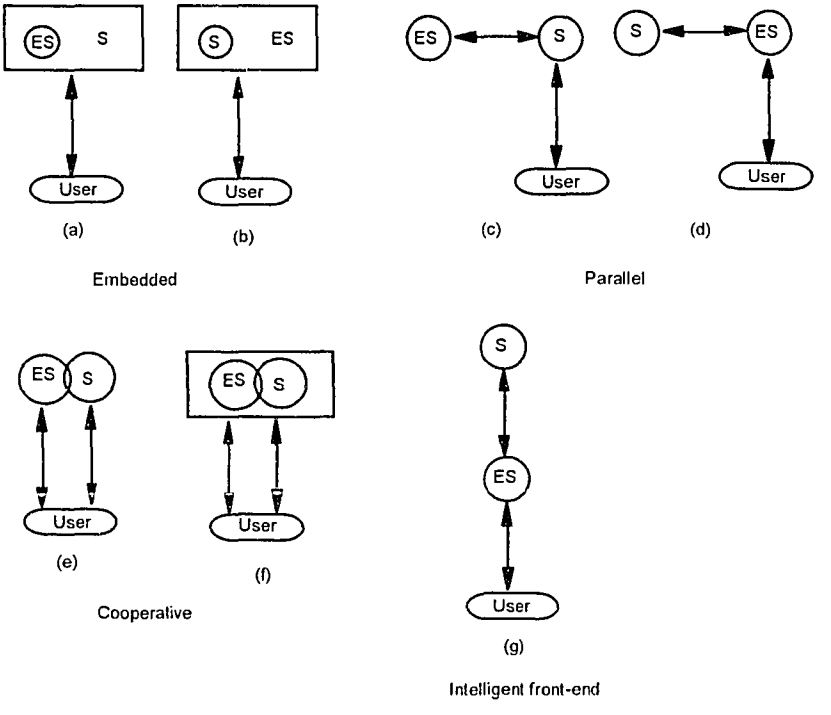


Figure 2.4. A taxonomy of combining ES and simulation as presented in [O'Keefe 86]

In another research, O'Keefe [O'Keefe 86] summarizes seven patterns and four types of AI and simulation integration (see Figure 2.4):

(1) *Embedding:* An expert system is embedded within a simulation model or vice versa (pattern (a) and pattern (b)). The expert system is conceptually part of the simulation environment.

(2) *Parallel:* Simulations and expert systems are designed, developed and implemented as separate software in parallel. A simulation model can interrogate an expert system (pattern (c)) or the expert system can execute and use the result from the simulation (pattern (d)). Pattern (c) is useful if a simulation is developed for a system where an expert system has already existed for part of the decision making in the system. Pattern (d) can be used to test an expert system in a simulation instead of on a real system, so that the development time and cost of the expert system can be reduced.

(3) *Cooperative:* AI and simulation share some data to fulfill certain tasks (pattern (e)) or they may be surrounded by a larger software system (pattern (f)). A good example of such a system is that an expert system can be a tool that helps the simulation modeling.

(4) *Intelligent front-end:* An expert system sits between a simulation package and the user, and generates the necessary instruction or code to use the simulation package.

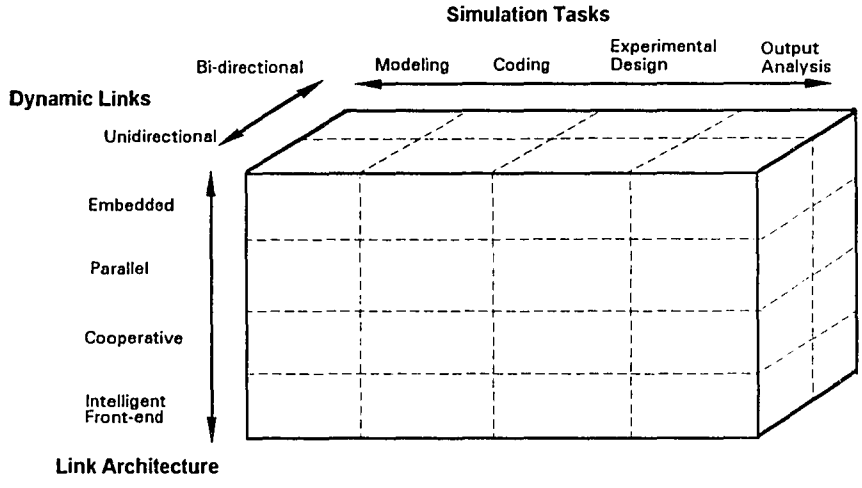


Figure 2.5. An extended ES and simulation model as presented in [Kwanjai 92]

Kwanjai and Wild [Kwanjai 92] extends O'Keefe's taxonomy to include two other dimensions: dynamic links and simulation tasks (see Figure 2.5). 'Dynamic links' refers to the actual interaction between an expert system and a simulation program. It can be unidirectional, in which data is transferred at some specified intervals and with only one direction, or bi-directional, in which data is transferred through a dynamic link back and forth at run time. Simulation tasks are associated with steps of simulation studies: from modeling and coding to experimental design and output analysis.

Many simulation expert systems have been developed to support different simulation tasks. The following are several expert systems that set good examples of AI simulation integration.

Hill and Roberts [Hill 87] illustrate an expert system prototype that helps students to develop simulation models. A knowledge base is constructed, which contains the expertise of common problems that students may encounter during simulation modeling with the INSIGHT simulation language. The prototype system is developed in

PROLOG. It helps students solve their simulation modeling problems through question-answer sessions.

Mellichamp and Park [Mellichamp 89, Park 90] have developed the Statistical Expert System for Simulation Analysis (SESSA) to provide support for the numerous statistical issues in simulation. It addresses as many as eighteen simulation analysis issues. Within each issue, the system helps users to identify particular methods for different situations. Once a method is chosen, a statistical package embedded in the system can perform the actual calculations on the input data. The system is implemented on a PC and has 172 rules.

Frankel and Balci [Frankel 89] describe the help system for the Simulation Model Development Environment (SMDE). The help system has two major components: (a) the Assistant Manager that offers an introduction to the SMDE, tutorials on how to use the tool, a glossary of terms, and a help-update facility, and (b) a tool specific help function provides the tool-implementor a set of routines to include in the application code.

Kreutzer [Kreutzer 90] discusses the Modeller's Workbench and the Modeller's Assistant. The Modeller's Workbench allows rapid prototyping of graphically animated queuing scenarios. The Modeller's Assistant is a production rules-based intelligent help system for the Modeller's Workbench. The system is implemented in SmallTalk on a Sun Workstation.

Wu [Wu 90] discusses the concept of an expert simulation system (ESS) combining simulation knowledge and domain knowledge into an environment which will automatically generate working simulation models. ESS is an expert system with an embedded simulation package. It includes: (1) a friendly user interface to input models

and data, (2) an automatic model generation, (3) a simulation execution, (4) an automatic simulation analysis, (5) a simulation model adaptation, (6) a help and interpretation facility, and (7) machine learning.

Wild and Pignatiello [Wild 91b] introduce a reverse simulation concept in which expert systems and simulation can complement each other to enhance simulation experimentation. Reverse simulation is a heuristic procedure which starts with a desired performance target value or a range of values and dynamically adjusts the system design to conform to these user-defined performance targets. The expert system and simulation package negotiate a dynamic bi-directional linkage between them to check with the user's goals and adjust simulation parameters. As output, reverse simulation provides information useful in determining initial feasible values for system design variables that serve as a starting point for subsequent performance evaluation and optimization.

2.3.4 Simulation environment

Simulation environment is a term used to describe a variety of architectures and products which support the simulation tasks by aiding different stages of the simulation life cycle. To some extent, such an environment can be viewed as a special CASE (Computer Aided Software Engineering or Computer Aided Simulation Environment) tool for simulation studies. Some simulation development environments cover every step in the simulation life cycle, while others may cover only part of the life cycle or are specific to certain application domains.

Henriksen [Henriksen 83] describes and integrates a simulation environment that covers the whole simulation life cycle. It includes a model design language, a model editor, an input preparation subsystem, a statistics collection definition facility, an experimental design facility, a program editor and compiler, and run-time support.

Balci and Nance [Balci 87, Balci 92] discuss a prototype of a discrete-event Simulation Model Development Environment (SMDE), that has been under development since 1983. It includes: (a) a cost-effective, integrated and automated support of simulation model development throughout its entire life cycle, (b) an improvement in model quality by effectively assisting in the quality assurance of the model, (c) increased project team efficiency and productivity, and (d) decreased model development time. The architecture of SMDE is composed of four layers: (1) hardware and operating system, (2) kernel SMDE, (3) minimal SMDE, and (4) SMDEs (see Figure 2.6).

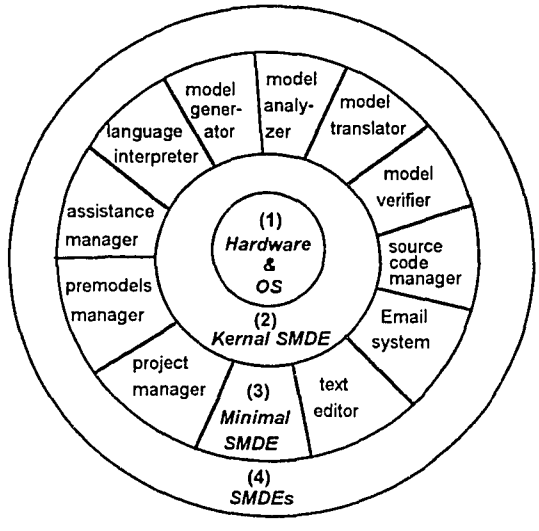


Figure 2.6. SMDE architecture as presented in [Balci 92]

Pflug and Prohaska [Pflug 90] introduce their Entity-Connection approach to modeling and simulation. They point out that the nature of simulation studies is such that the investigation of different model parameters, components, and structures is of primary interest over the specific results of a particular simulation run. Based on this observation, they conclude that a computer-aided tool for simulation should support easy model modification. The Entity-Connection approach is to support modularity and hierarchy in

modeling and programming. It also includes a graphical user interface of simulation models.

Graber, Ulrich and Bolay present their PetriNet-based object-oriented general purpose simulation system [Graber 90]. The system had an interactive graphical user interface to enable users to develop a model in a short time. The system has the same user interface for low level programming and for high level modeling. The user interface can be used by developers and by end users. Furthermore, the interface provides end users a fair chance to access lower level models, because all the models incorporate the same modeling philosophy.

Domain-specific simulation modeling environments are developed for specific applications. Because of the specific nature of an application, these environments can offer some special functions. An example of such a system is the NEtwork Simulation Testbed (NEST) [Dupuy 90]. It is a UNIX-based graphical environment for simulation and rapid prototyping of distributed networks and network protocols. NEST uses a different approach to simulation. It extends a network operating environment to support simulation modeling and efficient execution. This 'environment-based' approach offers a few important features: (1) simulation is integrated with the tools supported by the environment, (2) users need not develop extensive new skills or knowledge to use simulation, (3) standard features of the environment can be used to enhance the range of applicability. NEST uses client / server architecture that can serve multiple users interactively over communication networks.

2.3.5 Visual interactive simulation

Visual Interactive Simulation (VIS) is an important part of Visual Interactive Modeling (VIM), which includes both deterministic modeling, such as spreadsheet, and

dynamic modeling, such as simulation. VIM is a natural extension of Management Science and Operations Research. It combines an interactive interface, a visual display of computer-generated model status, and mathematical or symbolic models of a system to aid in decision making [Bell 91]. As a part of VIM approach, VIS produces a dynamic display of a system model and allows users to interact with the running simulation. It breaks the tradition of batch simulation methods and provides users with a visual and interactive simulation environment. VIS can be embedded into a Decision Support Systems (DSS) and Executive Information Systems (EIS) to support the decision making processes of non-technical managers and senior executives.

Bell and O'Keefe [Bell 87] suggest that a VIS environment should have three basic facilities: (a) *Visual Output*,(b) *User Interaction*, and (c) *Visual Input*. Among these three facilities, 'visual output' is absolutely necessary to visually describe a system, and 'visual output' plus 'user interaction' are necessary and sufficient. The majority of VISs have not provided sufficient support for 'visual input' so far [O'Keefe 87].

The major benefits of using VIS over traditional batch simulation methods are the following [O'Keefe 87]:

(1) *Selling:* VIS, particularly visual output or animation, is a tremendous aid to selling the simulation method, a simulation model or a specific solution. Users can quickly understand the model behavior and validate the simulation by following the dynamic display.

(2) *Gaming:* Using model determined interaction, user interaction can be incorporated into the model. Decisions that are too difficult to encapsulate in a model can be made by users. Gaming is particularly appropriate for complex

systems that are never allowed to reach steady-states due to a necessity for frequent management interventions.

(3) *Learning:* In addition to being used as an analysis tool, a VIS can be used by users to 'play' different scenarios of a system. The benefit to users is an increase in understanding of the system behavior and some information that can help to solve ill-structured problems.

Currently, there are no guidelines for VIS. New methods for statistical analysis are needed that include the user intervention factors. There are questions as to the efficacy of a VIS in simulation. Arguments have been made that in every simulation study, there comes a time when it is necessary to shut off the visual display and run properly designed statistics. There is a mistrust of the use of a VIS for experimental analysis and suggestions that using VIS for experimentation should be limited to professional users [Bell 91]. A study conducted by O'Keefe and Bell shows that although sometimes a more detailed analysis is required, a VIS is a good vehicle for simulation. They indicate that VIS animation is valuable, that confidence in decisions is warranted, and that the use of a VIS under a particular strategy leads to more efficient and better use of the model [O'Keefe 92].

The development of systems that provide VIM and VIS capability represent a major trend in the simulation area [Vujosevic 90]. Some general-purpose simulation packages, such as GPSS, SIMSCRTPT and SIMAN, have included a number of graphical features. They allow movement of objects in two dimensions or even three dimensions, animation of transaction movement in block diagrams, and dynamic statistical displays. Special-purpose VIS packages are primarily used in transportation scheduling and project management [Bell 91].

Hurrion, who proposed the concept of VIS in 1976, proposes a structure for an intelligent VIS that combines VIS and AI [Hurrion 91]. He asserts that without an expert system, a VIS can only react passively to user directions for pre-programmed conditions. An embedded expert system may add expertise to a VIS environment so that it will become participative along with users in search of an acceptable solution to the original problem.

2.4 Summary of Literature Review

This literature review provides support for the choice of techniques and architecture of HAT. While improving software quality is the focus of most computer scientists and MIS developers, upper CASE has drawn increasing attention because of the high probability of error and relatively low cost for error-correction in the early stages of the SDLC.

As one of the enabling techniques to improve CASE tools, hypertext has been used on several occasions in information systems to increase the user friendliness and improve document management. The non-linear structure and flexible store and retrieval capacity makes hypertext suitable for handling the complexity of IS development and provides both the computer professionals and users an easy access to design documents. A more friendly and easy-to-use CASE tool encourages more user-involvement and wide-spread use of software engineering methodologies such as JAD, DFD, ERD and so forth. As a result, more friendly CASE tools will eventually improve the quality of software products.

By embedding a simulation package within a CASE tool, the tool acquires dynamic model evaluation capacity which alleviates the bias of static evaluation. An expert system can be incorporated with a simulation package to aid the simulation

modeling and the simulation result explanation. There are several different architectures for expert system and simulation combinations. A simulation environment itself can be seen as a CASE tool that serves a special domain.

CHAPTER 3 METHODOLOGIES AND TOOLS

This research is interdisciplinary by nature and consists of four topics: software engineering, hypertext user interface, queuing network simulation, and rule-based expert system. Among these topics, software engineering is the fundamental one. In a broader sense, this research encompasses computer science, user behavior, Operations Research (OR) and Artificial Intelligence (AI) as indicated in Figure 3.1. HAT is implemented under Microsoft Windows with C++. MS Windows provides basic features of multi-window interface. The Dynamic Data Exchange (DDE) facilities of MS Windows serve as dynamic links among different system components. The C++ programming language provides a vehicle for object-oriented design and implementation.

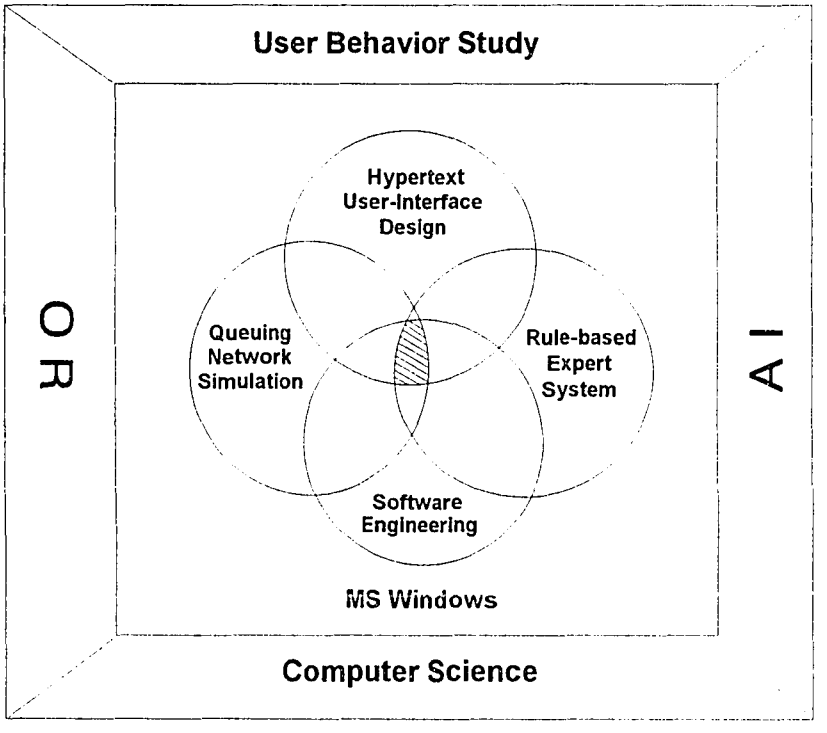


Figure 3.1. Areas involved in this research

3.1 Issues in User Interface Design

3.1.1 User interface design concepts

The user interface of a computer system should serve as both a connector and a separator of the system. The user interface design principles builds on the concepts of computer science, ergonomics, linguistics, psychology, and social science. Today's system designers are expected to apply these interdisciplinary principles to improve user satisfaction and productivity [Gerlach 91]. A typical interface design involves many decisions concerning the functions and objects to include, how they are labeled and displayed; whether the interface should use a command language, menus, or icons; and how on-line help can be provided.

From a design perspective, discretionary capacities and levels of expertise are the main distinguishing characteristics of users [Galitz 93]. Galitz summarizes these differences of users as:

(1) *Non-discretionary use:* Users in this group must learn to adapt to a computer, because this is the only way to get their job done. These users normally have technical backgrounds and are willing to invest time and effort in learning to use computers. They often have high motivation to use computer systems and can overcome the low usability of the systems.

(2) *Discretionary use:* Users of this group are more self-directed - not being told how to work but being evaluated on the result of their efforts. These users are office executives, managers who have been working without computers for years. They are neither willing to invest extra effort to learn computer systems nor are they interested in technical details.

(3) *Novice use:* Novice users are new to computer systems. They heavily depend on system features and facilities, such as menus, instructions, and help systems. These users prefer to have very informative feedback, simple tasks and tutorials to improve their system expertise.

(4) *Expert use:* Expert users rely on their experience and recall. They expect rapid system performance and less feedback. System efficiency is their primary concern rather than surface features of a system.

Table 3.1 illustrates computer users based on this taxonomy. The primary target of HAT is *novice users of non-discretionary use*, namely the users who have not much experience with information system analysis and are willing to learn structured systems analysis techniques. An extension to the architecture and concepts of HAT may result in computer systems that include some discretionary users who are interested in using structured techniques to model their day to day work.

Table 3.1. User classification

	Novice	***Expert***
Non-discretionary	computer operators, field specialists, secretaries,	computer specialists, system developers
Discretionary	office executives, managers of non computer department	system managers, project managers

Many studies have been conducted on user behavior in accomplishing specific tasks. The result of these studies are used to improve the cognitive processes employed in user interface design. Card [Card 91] proposes a three-stage user recognition cycle as the basic behavior for understanding the psychology of a user interface. A user will: (1) perceive the computer presentation and encode it, (2) search long and short-term

memory to determine a response, and (3) carry out the response through an action. A more elaborated seven-stage user interface model was proposed by Norman in 1986 (see Figure 3.2) [Norman 86]. Norman's model expands the memory stage to include mental activities, such as interpretation and evaluation of system responses, formulation of personal goals and intentions, as well as specification of action sequences.

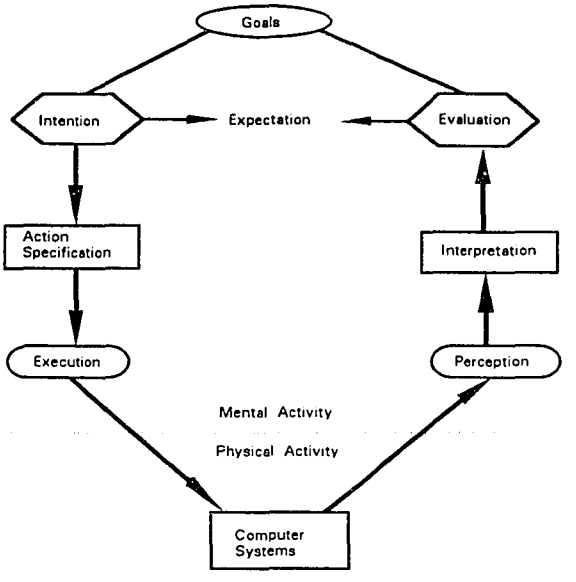


Figure 3.2. Norman's user interface cognitive model as presented in [Gerlach 91]

Gerlach [Gerlach 91] states that the goal of user interface design is to satisfy users and their perspective needs: signals must be perceivable, and responses should be within the range of a user's skill. He further states that the more important goal of user interface design is to empower the memory and cognitive capacity of users so that they can learn and reason the systems behavior. A human being is a complex organism with a variety of attributes that should be considered in user interface design. The attributes of particular importance are: perception, memory, visual acuity, skill and individual differences [Galitz 93].

3.1.2 User interface design methodologies

The design of a user interface still remains more an art than a science. An interface cannot be viewed as an 'add on' part that is developed in isolation. The methodologies for user interface design are part of the methodologies for interactive systems development.

Based on qualitative empirical observation of computer user interface developers, Hartson and Hix [Hartson 89] discuss the pros and cons of current software engineering methodologies for user interface design and conclude that user interface development naturally occurs in 'alternating waves' of two kinds of complementary activities: upward and downward where upward activities are synthetic, empirical and related to the end user's view; and downward activities are analytic, structuring, and related to the system view. These results suggest a 'star' life cycle for user interface development, as shown in Figure 3.3. This star life cycle, with evaluation as its center, supports iterative refinement and rapid prototyping. Because of its high interconnectivity, this model allows almost any ordering of development activities and promotes rapid alternation among them.

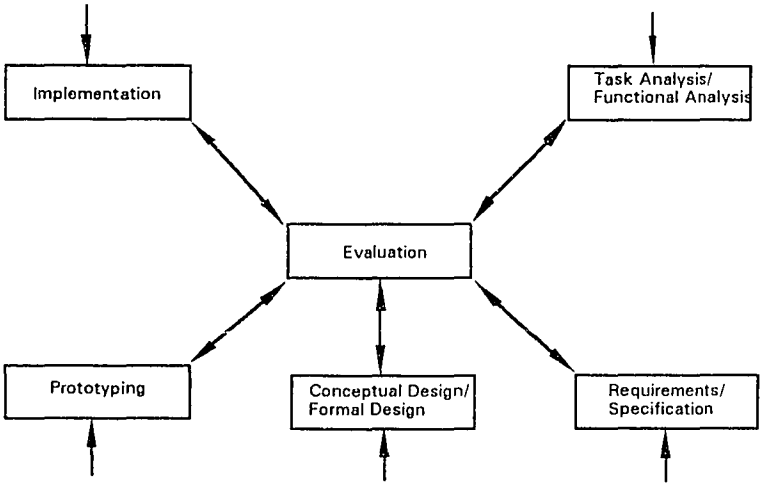


Figure 3.3. Star life cycle for user interface development as presented in [Hartson 89]

Fischer [Fischer 89] echoes Hartson's opinion by stating that the best paradigm for creating a user interface software is a communication model and a rapid-prototyping approach that supports the evolution of specification and implementation. Furthermore, Fischer points out that it is the human factor that distinguishes the user interface from other software. A computer system should include knowledge of the human factors in its user interface. Figure 3.4 is a knowledge-based human computer interaction model. The *explicit communication channels* in the model are graphical screen, windows, menus, pointing device, and audio/video input and output devices. The *implicit communication channels* are layers of knowledge structures that support the human-machine interaction.

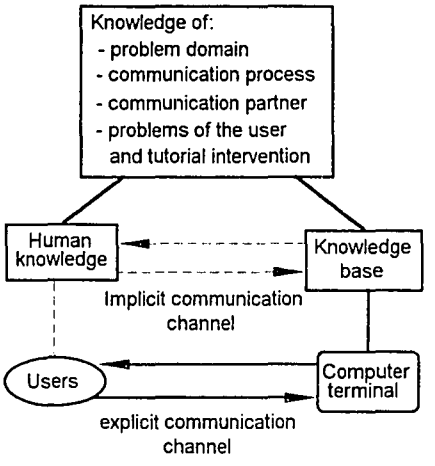


Figure 3.4. Knowledge-based human-computer interaction model as presented in [Fischer 89]

The issues of user interface design have great depth and subtlety. There are no concrete rules to guarantee a good design. Galitz [Galitz 93] proposes a set of guidelines for user interface development:

- *Consistency:* A system should look, act, and feel the same throughout. Consistent designs can reduce the requirement for human learning. Standard formats and screen layouts should be used to achieve the consistency.

- *Design tradeoffs:* Human requirements must always take precedence over machine processing requirements. When there is a conflict among different requirements, users' requirements should go first.

- *Flexibility:* A system must be sensitive to the different needs of its users. Flexibility is the ability to respond to individual differences. A flexible system should permit users to interact with it in a manner commensurate with their own knowledge, skill, and experience.

- *Complexity:* A system should minimize its complexity to perform required functions by hiding some information until it is needed. Uniformity and consistency of design will also simplify a system.

- *Closure:* A system should provide organized sequences of actions with a beginning, middle, and an end. Feedback should be available at the end of these sequences. Closure with its informative feedback provides users the satisfaction of accomplishment and a sense of relief.

- *Information load:* A system should be commensurate with the capacity of users and satisfy the users' information needs. Graphical and formatted displays can reduce users' information load as opposed to powerful commands and complex dialogues.

- *Control:* Users should control the interaction. All actions should be the results of user inputs and these actions can be interrupted and terminated be users.

- *Feedback:* A system should acknowledge all actions by immediate execution, change in status, confirm messages, or 'in progress' messages. Proper feedback will shape users' performance and instill confidence.

- *Recovery:* A system should permit commands or actions to be aborted or reversed.

- *Command language:* Command language should be logical, consistent, and flexible.

- *Error management:* A system should have error prevention, detection, and correction capacity. Good error management can save users' time and frustration and improve user confidence.

- *Response time:* A system's response time should match the speed of human thinking processes.

- *Guidance and assistance:* A system should provide on-line documentation that supplements hard copy documentation and *help* facilities.

HAT is a user-oriented system. Its cardinal goal is to help users learn and use structured systems analysis. The design of the HAT user interface observes the above guidelines.

3.1.3 User interface tools

User interface software is often large, complex and difficult to debug and modify. As user interfaces become more friendly, they become harder to create. The increasing complexity comes from the 'easy-to-use' features of modern systems, such as elaborate graphics, many ways to give the same command, control of many input devices, and mode-free interactions. In some applications, 40-50% of the code and run-time memory are devoted to interface functions [Myers 89]. It is imperative that computer aided tools and interactive design strategies are used for user interface designs.

User interface tools come in two general forms: *user interface toolkit* (UIT) and *user interface development system* (UIDS). A UIT is a library of interaction techniques, which provide ways to use physical input and output devices. Examples of interaction techniques are menus, scroll bars, buttons and cursors. A UIDS is an integrated set of tools that help programmers to create and manage many aspects of a user interface.

A UIT is often a cluster of application programs that supports interaction techniques. It does not provide enough support for the design of interfaces or the sequence of dialogue control. A system developed with a UIT often takes more time. A UIDS helps the designers combine and sequence different interaction techniques and provide interactive access to the techniques. There are language-based and graphical-based UIDS. Some UIDS support automatic creation of user interfaces.

In the MS Windows environment, Microsoft provides a set of Windows programming utilities, called the System Development Kit (SDK). These utility programs can be used as a UIT to develop Windows applications. In addition to the SDK, Borland provides an interactive Resource Workshop for interface design. The Resource Workshop is a special UIDS for menus, icons, dialogue boxes, and bitmaps of Windows. Users can define a user interface component in the Resource Workshop graphically and the system will convert the resource description to computer code automatically. Other Windows Programming environments, such as Visual BASIC and Visual C++ of Microsoft, also offer visual programming capacity.

HAT uses another UIT package – ObjectGraphics, in addition to the functions provided by MS Windows and Borland C++ programming environment. ObjectGraphics is developed by the Whitewater Group and Application Vision Inc. It is built on top of Borland OWL (Object Windows Library) and Microsoft SDK. It provides a set of

object-oriented classes for graphical object manipulation. HAT inherits some ObjectGraphics objects and develops its own objects for DFD and ERD graphical interface design.

Table 3.2. Pros and cons of user interface design tools from [Myers 89]

Advantages	***Disadvantages***
• Design can be rapid prototyped before coding. • Quick feedback for modification. • Easier to investigate different user interface styles. • More efficient use of resources, because tools can be used in many projects and many times. • Easier for field specialists involved in design. • Code will be better structured and more modular. • Code will be more reusable. • Higher reliability of user interface	• Language-based tools are difficult to use and the specifications are hard to understand. • Not enough functionality is offered. • Tool are often not portable. • Interface evaluations are not available. • Very hard to build tools. • Difficult to separate user interface from the application. • Designers are unwilling to accept new tools.

Myers [Myers 89] summarizes the pros and cons of current user interface tools (see Table 3.2). It is expected that in the 90's the user interface technologies are converging to Speech, Image, Language and Knowledge (SILK) capacities [Marcus 91]. The development of enabling technologies in areas of fast graphical processor, 3-D hardware and software, hypertext and hypermedia, and virtual reality, will solve some of the user interface problems we are facing now.

3.1.4 Evolutionary development strategy for hypertext applications

The development of a hypertext application requires a good understanding of the structure of the application. Again, software engineering principles should be applied. This often means that applications have to be well-structured, which in many cases need a hierarchical skeleton. The links that correspond to the hierarchy (e.g. vertical links from one DFD to another) are easy to present. However, links that correspond to other relationships (e.g. horizontal links from a data store to its data node) are more difficult to manage, although they are necessary to constitute the hyperlink network.

There are many text books that claim to provide for structuring and managing methods of hypertext application development [Martin 90b, Horton 90]. However, these methods only provide guidance at a coarse level and have rather limited support for teamwork [Oinas-Kukkonen 93]. Oinas-Kukkonen combines the evolutionary development method in software engineering with a hypertext system design and proposes a model for evolutionary hypertext application development shown in Figure 3.5.

This model takes advantage of the easy-to-use tools in many hypertext development environments (Toolbook, Plus, HyperCard, etc.) and emphasizes the use of prototyping techniques. Prototyping is of value for creating hypertext applications and making not only one but many evolving prototypes during the development is helpful. The evaluation of these prototypes by developers and end users is of great benefit to the design process. This model encourages 'quick-and-dirty' prototyping at many phases of a development process. However, these prototypes should be thrown away, as the author suggested, and implementation should proceed from a 'fresh start' to prevent hidden obstacles for implementation retained in the prototypes [Oinas-Kukkonen 93].

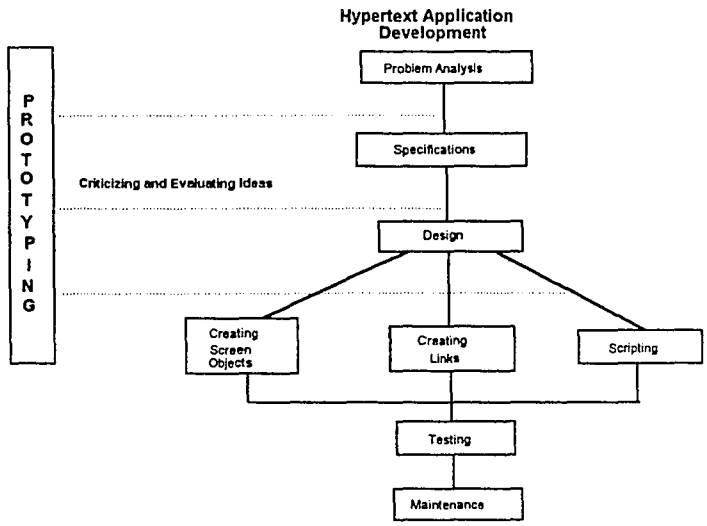


Figure 3.5. Evolutionary development strategy for a hypertext system

3.1.5 The DEXTER hypertext reference model

Halasz and Schwartz [Halasz 94] describe the DEXTER hypertext reference model as an attempt to capture, both formally and informally, the important abstractions in a wide range of hypertext systems. The goal of this model is to provide a basis for comparing systems as well as for developing interchange and inter-operability standards.

Figure 3.6 shows an overview of the DEXTER reference model. A hypertext system is divided into three layers: the *run-time* layer, the *storage* layer, and the *within-component* layer. The run-time layer provides tools for users to access, view, and manipulate a hypertext network. The storage layer focuses on the mechanism by which link and non-link components of a hypertext system are 'glued together' to form hypertext networks. The within-component layer concerns the contents and structure *within* the components of a hypertext system.

A crucial piece of the DEXTER Model is the interface between the storage layer and the within component layer that addresses locations or items within the contents of an individual component. This interface is known as *Anchoring*.

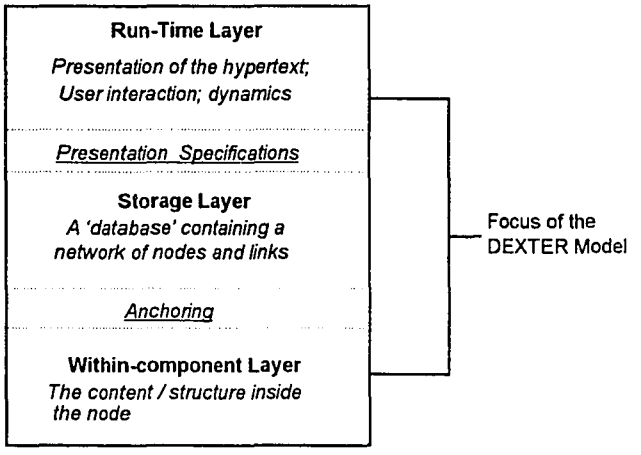


Figure 3.6. An overview of the DEXTER model layers as presented in [Halasz 94]

Another important part of DEXTER model is the interface between the run-time layer and the storage layer, known as *presentation specifications*. Presentation specifications are a mechanism to present information about how a hypertext component network is to be presented to users.

Even though the development of the DEXTER Model is still in its very early stages, the model is far more powerful than any existing hypertext system [Gronbaek 94]. It can provide the basis for developing a comprehensive standard for interchanging hypertexts between different systems.

3.1.6 User interface evaluation

With the new GUI development techniques and environments, the temptation to move applications to GUI increases. However, user interface design is often more difficult than it looks. Evaluation of a GUI user interface is also a very expensive

process. It requires the support of cognitive scientists, such as psychologists and graphic design specialists. The evaluation often involves designing and carrying out a series of user experiments and statistical analyses. A complete evaluation is economically possible and sensible only for large projects. The methodologies for complete user interface evaluation are beyond the scope of this project. However, some simple evaluation techniques can be carried out for a project like HAT:

- Questionnaires: Survey users with a questionnaire to collect information about what users thought about the interface.
- Observation: Observe user working with the interface and record their reactions to screen displays and window lay-outs.
- Video recording: Record user sessions and analyze the sessions for hand movement, eye movement, facial expression, and etc.
- Embedded coding: Include a piece of software code that collects information about the usage of facilities and errors.
- Comment insertion: Provide facilities to allow users to feedback comments to system designers.

3.2 Object-Oriented Simulation and YANSL

3.2.1 Advantages of object-oriented simulation

Object-oriented programming is a design and programming discipline that focuses on the objects that make up the system rather than on functions of the system. Object-oriented simulation (OOS) uses object-oriented principles for simulation modeling and program design. The merit of OOS is that it conforms to the notion that

the world is composed of 'objects'. For example, a hospital can be seen as an assembly composed of many 'objects': doctors, nurses, medical records, and X-ray machines. Objects can also describe things that are not physically presented, such as a concept, a record in database, etc. As a result, OOS language and modeling have great intuitive appeal [Lomow 91].

OOS preserves all the features of an object-oriented system, such as data encapsulation, inheritance, abstraction, dynamic binding, software reuse, etc. Bischak and Roberts [Bischak 91] point out that the most important virtues of OOS are reusability and modularity. Because of the inheritance feature of OOS, model designers can create their own version of simulation objects by defining new features and re-using features inherited from the parent objects. With an OOS language, users do not have to try to match what they want to do in a simulation model to the limited number of constructs available in the simulation language. The OOS modularity allows all the information about an object to be held in one place, which means that changing an object or modifying its behavior can be easily achieved.

Brischak and Roberts foresee four areas in which the object orientation has special potential [Brischak 91]:

- *Graphical presentation:* Objects in an OOS tend to represent 'real-world' entities and they 'encapsulate' these real-world behaviors. Graphical representation can have almost one-to-one correspondence with the objects in the simulation model. Furthermore, during the execution, such a correspondence can produce a very convenient basis for animation.

- *Combination of simulation and AI:* Objects encapsulate their functionality and that functionality can include 'intelligence'. OOS program can exhibit learning optimization ability by embedding AI algorithms into simulation objects.

- *Parallel execution of simulation:* Because of the encapsulation of information needed for an object, individual objects can be assigned their own processors to execute their behaviors in parallel.

- *Possibility for 'simulation software engineering':* The notion that users can build their own simulation objects gives rise to the possibility of simulation software engineering. Through object-oriented technology, a new category of simulation professionals may emerge that develop simulation tools for simulation application engineers who use simulation to solve real-world problems.

3.2.2 YANSL - an object-oriented simulation package

YANSL was developed by Professor S. D. Roberts and his students at North Carolina State University [Joines 92]. It uses C++ to develop a cluster of classes for general purpose discrete-event simulation. Figure 3.7 shows the class tree of YANSL.

There are over forty classes in this class tree. *SimObject* and *Link* are the most fundamental classes from which other classes of simulation objects are derived. Several link-list classes are created to manage nodes in a simulation model, simulation events, and simulation statistics. The class tree also includes a random number generator and a distribution generators necessary to generate random streams. Because of the object-oriented nature, YANSL is not limited to the classes described in this class tree. It is extendible for different needs. For example, if a new distribution is needed for a simulation, it is easy to add another class as sub-class of *Random* that generates the

distribution stream without changing any other classes. Similarly, new classes can be added to describe special simulation nodes or collect specific statistics.

C++ provides different ways to create instances of objects statically or dynamically at run time. YANSL provides a toolbox for simulation model designers that contains building blocks for simulation modeling. Simulation model designers view models as a network of elemental queuing processes. Building a simulation model requires a designer to select from the pre-defined objects simulation toolbox and integrate these objects into a network. Because of the data encapsulation, the designer does not have to know the objects' internal structures to connect two objects. The designer can use the concept of source nodes, resources, queues, servers, and sink nodes to build an inter-connected network model and run this model on the framework provided by YANSL.

HAT uses YANSL as the simulation kernel for dynamic DFD evaluation. Some add-on features are added to the original YANSL classes to make them suitable for the Windows environment.

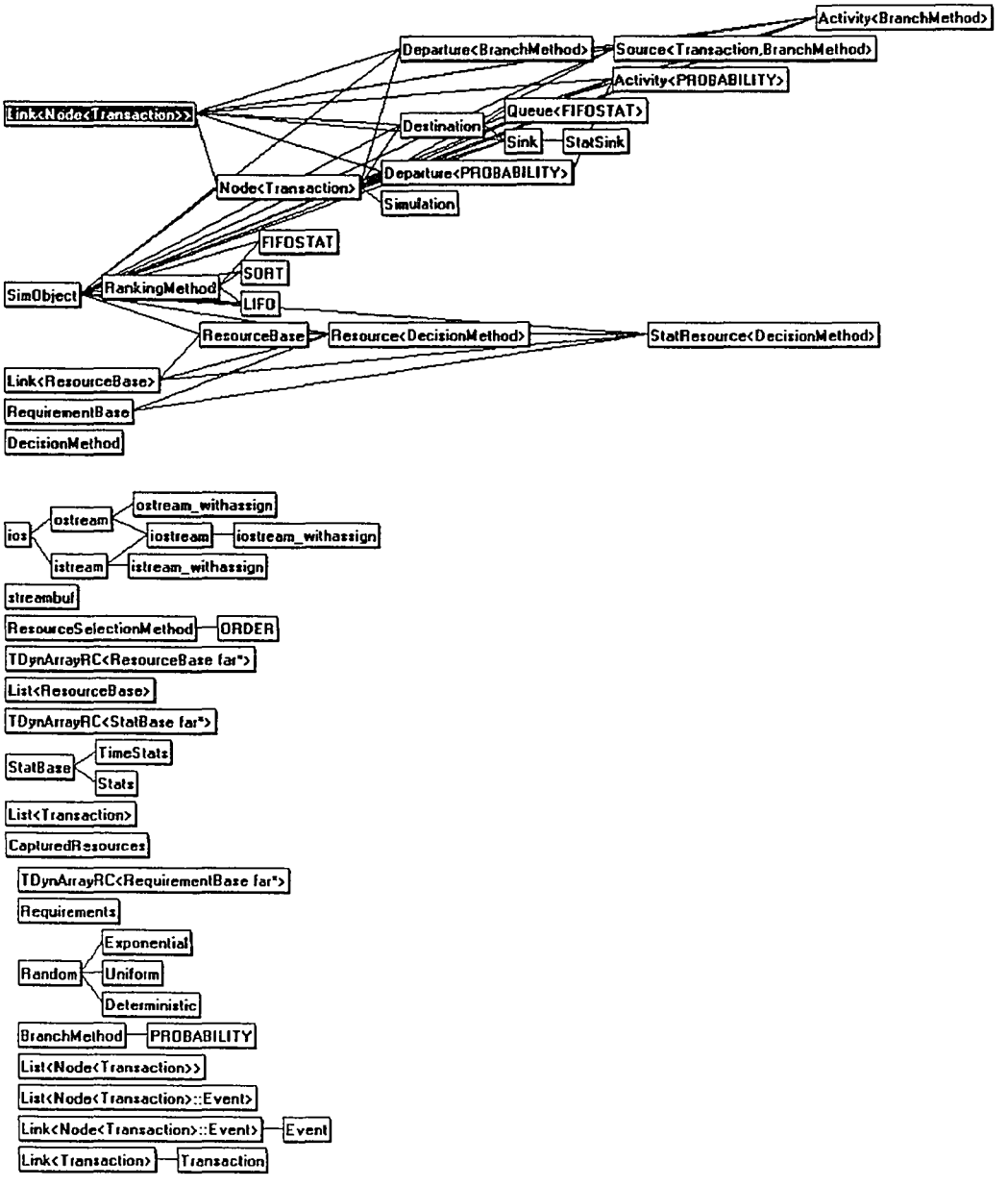


Figure 3.7. Class structure of YANSL simulation language

3.3 Rule-Based Expert System and M4

3.3.1 Basic concepts of expert systems

As one of the most active branches of Artificial Intelligence, expert systems use human knowledge and experience to solve problems that require special expertise. Expert systems enhance productivity by making this expertise available to inexperienced users, helping them make decisions and solve problem effectively. By using expert systems to distribute the decision making and problem solving expertise, an organization can optimize its resources, reduce its cost, and become more competitive.

An expert system is normally composed of a knowledge base, an inference engine, knowledge acquisition facilities, explanation facilities, and a user interface. The knowledge base of a rule-based expert system is constructed with facts and if-then rules. An inference engine is a software system that locates knowledge and infers new knowledge from the base knowledge. Two fundamental methods are often used by inference engines: *backward chaining* and *forward chaining. Backward chaining* is a top-down reasoning process that starts from the desired goal and works backward toward the requisite conditions. *Forward chaining* is a bottom-up reasoning process that starts with known conditions and works toward the desired goal.

Applications of expert systems exist in many areas, such as manufacturing, finance and administration, data processing, management information systems, engineering, and training and education. The types of applications fall into five distinct areas: analysis, planning, design, selection and diagnosis [Keller 87].

- *Analysis applications:* Analysis or data interpretation involves monitoring data streams and interpreting trends and other factors.

- *Planning applications:* Skeletal planning involves selecting a number of sub-plans from a library of possibilities, integrating them into a generalized plan, and then tailoring the plan to the current specific situation.

- *Design applications:* Skeletal design involves choosing one of a number of design schemes, tailoring them to meet design constraints, and producing a design document.

- *Selection applications:* Catalog selection involves mapping, or translating, from terms that a user understands to features of items in a catalog database.

- *Diagnosis applications:* Fault diagnosis involves providing advice on what might be wrong with an entity, that can be a piece of equipment, an organization, or a person.

3.3.2 The M4 expert system

M4 is a general-purpose rule-based expert system developed by Cimflex Teknowledge. It works in both DOS and Windows environments. The M4 expert system kernel can be embedded into other C or C++ based applications. In the Windows environment, M4 also provides a ready-to-use library that can be integrated with Visual BASIC applications. Figure 3.8 shows the structure of the M4 kernel. It contains a forward and backward chaining inference engine, knowledge base management facilities, symbolic pattern matching mechanisms and an M4 script language interpreter. The kernel is structured by different modules. Users can adjust environment parameters, modify or replace modules, and fine-tune the expert system kernel for special applications.

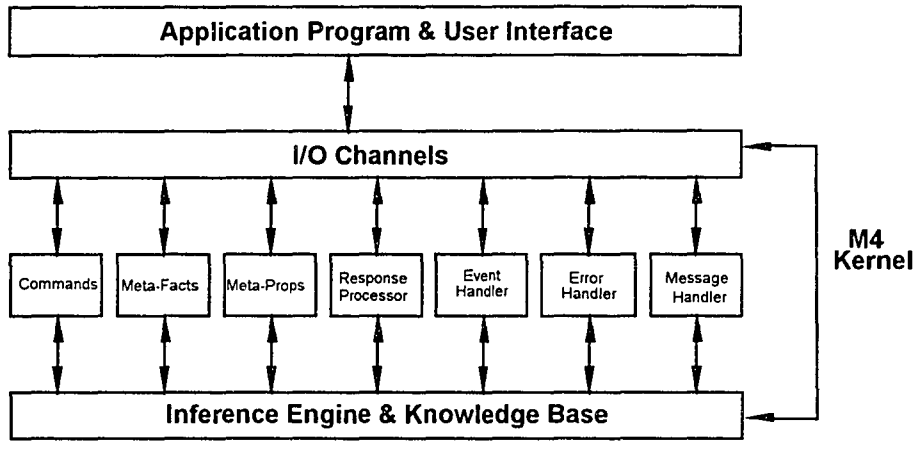


Figure 3.8. The kernel structure of M4

The activities of inferencing and interaction with the user interface are controlled by major modules in the kernel. These modules include:

- *Inference Engine:* Performs forward and backward chaining inference with full symbolic pattern matching.
- *Response Processor:* Prompts users with questions, collects the response, and validates the result.
- *Error Handler:* Handles errors that occur while M4 is running, including language parsing errors, variable errors, and run-time errors.
- *Message Handler:* Processes messages generated by M4 commands, meta-facts, and meta-propositions.
- *Event Handler:* Processes events generated by the inference engine.
- *I/O Channel:* Connects the M4 kernel and communicates with the user interface through a set of different types of input and output.

M4 has its own knowledge representation language and language interpreter to describe facts, rules, confidence factor of inference, and question / answer dialogues of a consulting session. Therefore, a rule base can be directly loaded into the M4 inference engine. Since the M4 expert system kernel takes care of the details of knowledge processing, the major tasks for expert system development become knowledge acquisition and knowledge representation with the M4 knowledge representation language.

HAT uses M4 to build its intelligent help subsystem for simulation modeling and simulation result explanation. All the basic M4 functions are preserved in the subsystem which is implemented with Visual BASIC in Windows.

3.3.3 Procedures for expert system development

The development of an expert system requires similar steps and quality assurance as any other software. In addition, expert system developers have to consider specific problems of knowledge domain definition, knowledge acquisition, and knowledge representation. Ignizio [Ignizio 91] summarizes general procedures of expert system development as shown in Figure 3.9. This model assumes that the knowledge acquisition and knowledge representation methods have been defined and developers know what an expert can do and what an expert cannot do.

Simulation modeling and simulation result interpretation require a good understanding of simulation modeling, statistics, and stochastic analysis. The domain knowledge is specific and has been clearly documented. Knowledge for such an expert system is available from simulation experts and simulation literature. HAT takes advantage of the M4 expert system facilities to build the rule-base for the simulation expert system and execute the rules on the M4 expert system kernel.

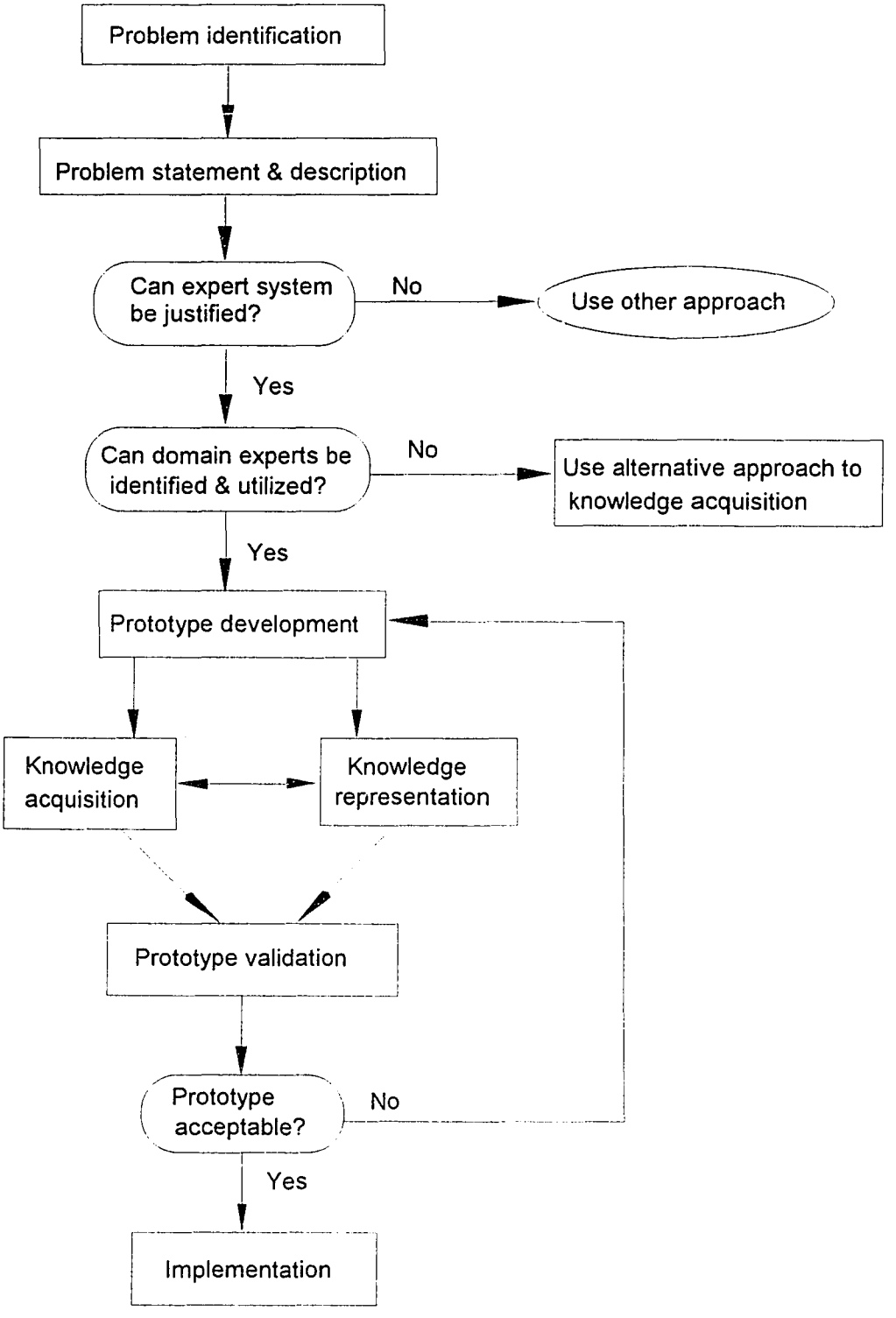


Figure 3.9. A generic procedure for expert system development

3.4 Dynamic Data Exchange and Object Linking & Embedding Techniques

Dynamic Data Exchange (DDE) is a MS Windows data exchange technology developed by Microsoft for user-independent dynamic data transfer among Windows applications. DDE is also the foundation of another important feature of Windows environment – Object Linking and Embedding (OLE). DDE and OLE are the major building blocks in the Microsoft vision of next generation computing. The concepts of DDE and OLE have essentially altered the way users think of software and provide a powerful tool for software integration.

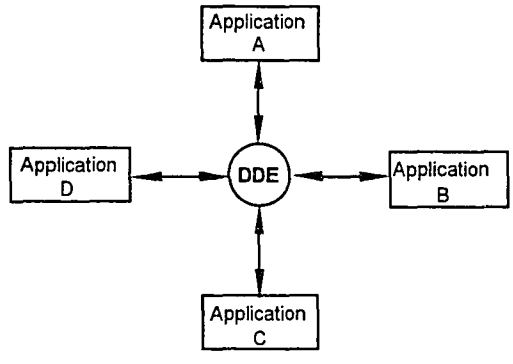


Figure 3.10. DDE as an information hub to connect different applications

DDE is a form of inter-process communication that uses shared memory to exchange data among applications (see Figure 3.10). Applications can use DDE for one-time data transfer (passive connection mode) as well as for on-going exchanges in which the application sends updates from one to another as new data become available (active connection mode). DDE is more powerful and flexible than a clipboard. A clipboard operation often requires user intervention, such as 'cut' and 'paste', and does not hold the connection after data transfer. DDE maintains the communication link and have the capacity to transfer multiple data items as long as the two parties involved in the communication want to keep the link. DDE does not need user intervention during data

transfer. It is fully programmable and flexible to allow any Windows application to participate in communication as long as it follows the DDE protocols.

DDE follows the client-server (destination-source) communication pattern. Before any data transfer, a client (destination) specifies the requested service attributes (service name, service topic, etc.) and initiates a DDE link. If the requested server (source) is active, it will check the request from the client and reply with a positive or negative acknowledgment. Once the DDE link is setup, data can be transferred in a 'passive' connection mode or 'active' connection mode from the server to its clients. In addition to the passive and active connection mode, 'poke' mode allows a client to pass a short message to its server. The DDE link will keep open until either the client or the server requests for disconnection. Multiple DDE client-server relationship are allowed, in which a client may have multiple servers and a server may serve multiple clients.

As a superset of DDE, OLE has potential to support higher level communication between two Windows applications. OLE defines 'packages' and 'verbs' to define embedded objects and the object actions. An OLE object can be an 'alien resident' in another application as a special OLE 'package'. When activated (normally by double-clicking), the OLE objects execute its 'native' application and perform special operations defined by its 'verbs'. OLE techniques provide an easy way to connect two Windows applications. The new Microsoft OLE 2.0 moves the original OLE concepts further to include basic OLE functions, OLE Automation, and OLE Control [Pleas 94]. It is believed that DDE and OLE will be a norm for software integration in Windows.

3.5 Object Manager and Object Database

A data repository is the center of a CASE tool. Strong [Strong 90] reviewed the major characteristics that a CASE environment requires for database support. These requirements are:

- *Support flexible data type:* The database should support a variety of numeric values and text strings up to some reasonable length.

- *Support metadata:* Metadata is a description of another set of data. A database often uses metadata to structure its manipulation of data. It is also useful to CASE tools as a guide to the handling of data in a data dictionary.

- *Query language:* CASE tools require that all data is reachable. Models and their descriptions should be easily accessible through a SQL-type query mechanism.

- *Integrity and internal consistency:* CASE tools need to keep track of the evolution of the design documents, programs, and program fragments. The database has to maintain different versions and alternatives of designs, and keep consistency at the same time.

Since HAT is an upper-CASE tool, not all these requirements are applicable to HAT. In addition, HAT is developed with object-oriented concepts. Object-oriented operations, such as persistent object storage and retrieval, are critical. In an object-oriented system, the entities in memory are viewed as objects, and so are the files on the peripheral storage. On top of the traditional bit-stream, record, file concepts, a higher level management needs to be built - the object-oriented management that provides service to store and retrieves information by objects. Persistent object storage and retrieval are different from ordinary binary-based or ASCII-based files, in which all the

attributes and links with other objects are preserved. Applications view the object-oriented persistent services as an object-based storage where they put and get objects regardless of how the objects are stored on peripheral devices.

3.5.1 A file-based object-oriented storage service – Tools.h++

Tools.h++ is a cross-platform C++ class library developed by Rogue Wave, Inc. It supports complicated object structures and object operations that reduce the burden on application programmers. Tools.h++ provides file-based object-oriented persistent storage and retrieval so that objects can be saved to disk and restored in a new address space, even on a different operating system. Other Tools.h++ functions include:

- *Smalltalk-like collection classes:* A complete library of collection classes, modeled after *Smalltalk* programming environment: Set, Bag, OrderedCollection, SortedCollection, Dictionary, Stack, Queue, etc.

- *Template based classes:* A complete set of collection based on C++ *templates*: single- and double-link lists, stacks, queues, has tables, sets, dictionaries, etc.

- *Generic collection classes:* As an alternative to template classes, generic classes provides similar services to a non-template programming environment.

- *String and character manipulation:* A string operation class provides operators and functions to manipulate character strings, including concatenation, comparison, indexing, I/O and many other functions.

- Other services including time and date handling classes, B-Tree disk manager, error handling, etc.

HAT uses Tools.h++ as the foundation to build its data repository as a storage and retrieval subsystem. The services provided by the Tools.h++ class library have greatly reduced the complexity of the implementation.

3.5.2 An object-oriented database – RAIMA Object Manager

Tools.h++ is limited to file-based services. Objects in different files cannot be shared. Although, Tools.h++ is sufficient for the preliminary implementation of HAT, object-oriented database services are required for large amounts of data in a multi-user environment where data sharing is fundamental. The RAIMA Object Manager is an object-oriented database developed by Raima Corporation. It is a C++ class library which includes the following features:

- Uses RAIMA Database Manager (a database on which the Object Manager operates) as the engine for its DBMS and makes use of the Database Manager's direct access techniques.

- Provides support for multi-tasking, and for incremental opening and closing of databases. Two databases of the same type can be used simultaneously.

- Automatic concurrency management enables multiple complex database navigation without disturbing the currency of each navigation.

- Allows programmers to add persistence to objects. The methods of a storable class that implement persistence can be either automatic or programmer controlled. This simplifies the storage and retrieval of objects and automatically maintains relationships between objects. Object-to-object relationships are implemented by three separate access methods: *Network database model, Relational database model, Direct sequential access.*

- Allows programmers to define container classes to manage arrays of storable objects. Once the databases and classes are correctly defined, the programmer no longer needs to be concerned about where an object came from or where it is located.

- Another Raima product, QUERY, can be used in conjunction with Object Manager to perform SQL-based queries on your database.

3.6 Summary of Tools and Methodologies

This chapter reviews the methodologies and tools used for the hypertext user interface, object-oriented simulation, expert system, DDE data exchange and object-oriented data storage. This chapter shows that the HAT implementation is feasible. The goal of integrating hypertext, simulation and expert system with software development is not only conceptually feasible, but also practical under the Windows environment with object-oriented techniques.

CHAPTER 4 THE SYSTEM ARCHITECTURE

As has been illustrated in Chapter 3, the concepts of HAT are feasible with the combination of existing techniques and programming environments. This chapter focuses on an architecture to implement these concepts. The discussion focuses on the Windows environment and the C++ programming language. In addition, Visual Basic is used to build a Windows-based shell for the M4 expert system.

The architecture follows object-oriented conventions and emphasizes the concepts of abstraction, data encapsulation, inheritance, polymorphism, and dynamic binding. The Coad-Yourdon method [Coad 90, 91] is used to describe the classes and objects. This method uses two major structures, *Classification Structure* and *Assembly Structure*, to describe object classes. Appendix A is a brief review of object-oriented analysis and design concepts as well as the Coad-Yourdon method.

An evolutionary development strategy coupled with the prototyping method is used for the user interface design. Because of the HAT project, it was difficult to pre-define what should be included in the system, what the user interface should look like and what is achievable for a given situation. Prototypes are very helpful to determine these factors and evaluate different alternatives for the implementation.

Figure 4.1 shows an exploratory model for developing a new system [Griggs 89], which reflects the process of HAT development. The initial concept creation phase consists of the formation of an idea and a researcher's introspection of his own experiences and feelings about a problem domain coupled with existing research. The second phase, trial and error construction, consists of developing computer software that encapsulates the researcher's concepts. The 'trial and error' aspect of the process reflects

the fact that the move *from concept to code* involves a series of iterations. It may be discovered that part of the original concepts may not be feasible with given constraints.

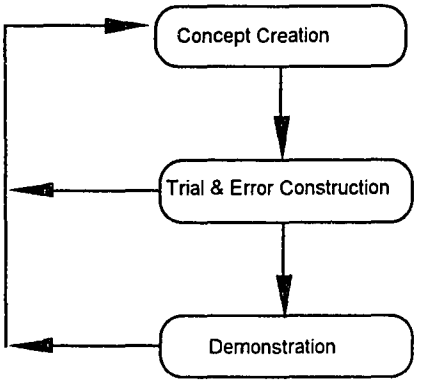


Figure 4.1. An exploratory model to develop new systems

4.1 The Software Architecture for HAT

The central theme of HAT is the integration of existing techniques to improve systems analysis. Since so many elements are involved in the system, it is wise to organize the system into several loosely-coupled, autonomous subsystems or objects. Therefore, each of the systems has the freedom to decide its own structure, processing methods and implementation strategies. Within such an integration, controls from one subsystem to another should be minimum to guarantee data encapsulation and prevent unnecessary ripple effects and data contamination. On the other hand, the connections for data and messages should be smooth and sufficient to link the subsystems into an integral whole.

Figure 4.2 shows the architecture of HAT integration. Four subsystems are presented: (1) the user interface, (2) the data repository, (3) the dynamic evaluator, and (4) the expert system. Each subsystem has the liberty to choose its own implementation strategy and environment. Since the user interface is the front-end to users and requires

frequent access to the data repository, it is directly coupled with the data repository as a single application, so that graphical objects and hyperlinks, and other objects created during systems analysis can be stored and retrieved effectively. On the other hand, the dynamic evaluator and expert system are both very complicated systems and are invoked only at discrete points of systems analysis. HAT allows these two subsystems to be independent from the user interface and the data repository as separate applications. This loosely-coupled architecture makes the subsystems maximize the choice of different alternatives and minimize the complexity of control. As a result, DDE data links are the only channels that connect the dynamic evaluation and expert system with the user interface.

For the user interface subsystem, visualization and interactivity are the primary concerns. Graphical user interface techniques are used to focus on an easy to use interface design. Since HAT is intended to be a front-end CASE tool, basic systems analysis tools and user friendliness are fundamental.

The data repository is designed for HAT to handle different objects and the hyperlinks among them. It requires flexible data structures, persistent object-oriented storage-retrieval capability and easy access from the multiple-window user interface. The data repository is also responsible to keep data consistency and manage a central log of objects that have been created.

For the dynamic evaluator subsystem, simulation modeling and simulation execution are critical. Calculation efficiency and accuracy are a top priority. More efficient programming languages and fully tested simulation algorithms should be used. The HAT architecture allows the simulation subsystem to inherit and reuse existing

simulation packages with little constraints from the design of either the user interface or the expert system.

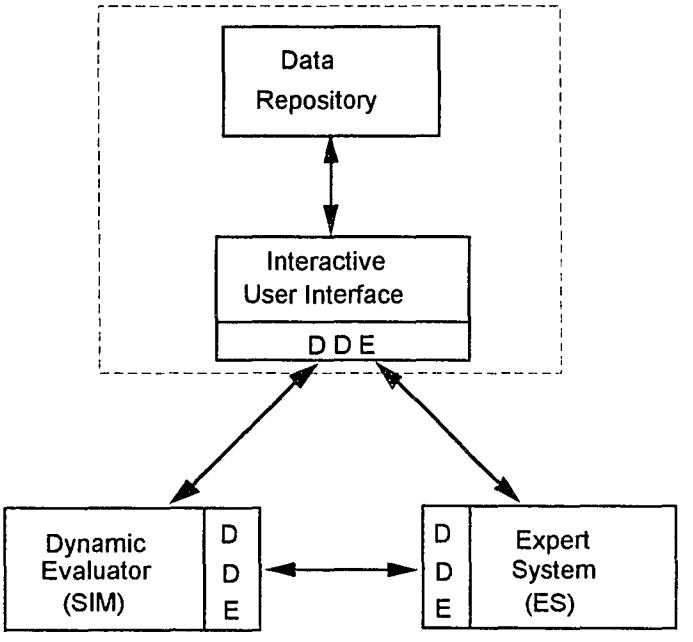


Figure 4.2. The architecture for HAT integration

The major concern of the expert system is the efficiency and effectiveness of the inference engine and rule-base management system. A general purpose rule-based expert system is sufficient for the needs of the simulation expert system in HAT. Like the case of the simulation subsystem, it is not necessary to create a specific expert system. A Windows-based expert system can be incorporated into this triangular architecture and monitor the simulation execution.

This architecture takes advantage of the Windows environment as well as existing simulation systems and expert systems. It observes the principle of software modularity and encourages software reuse. An architecture based on this triangular structure with improved DDE data links and a more sophisticated expert system can be used for reverse simulation and other simulation environments as well [He 94b].

4.2 The User Interface Subsystem

The user interface subsystem is a multiple windows application consisting of a DFD Editor, ERD Editor, Hypertext Editor, DFD Browser, DFD Descriptor and windows for project and data dictionaries.

Since HAT is targeted at novice, non-discretionary users, simplicity was a guiding principle in the design of the user interface. All the functions can be accomplished by simple 'point', 'click', 'drag' and 'drop' operations, that are standard for most Windows applications. HAT also simplifies standard CASE functions so that novice users can quickly grasp the principles of systems modeling. The design also enforces basic rules of structured analysis to help users create correct system models.

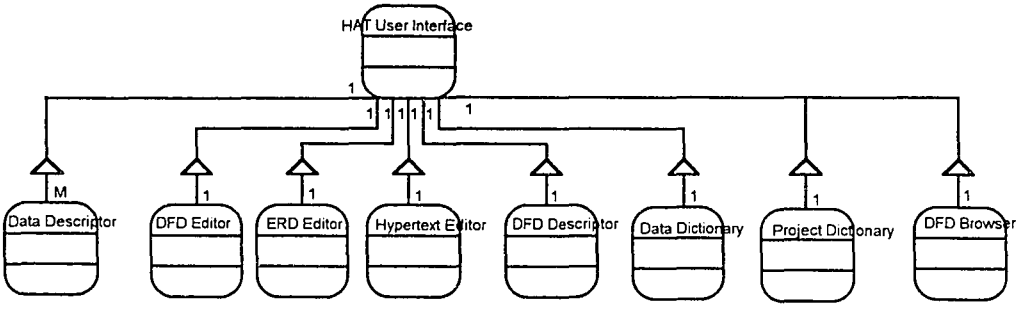


Figure 4.3. Assembly structure of objects for the user interface

The user interface is an object that inherits basic features of multiple-window management from Microsoft MDI (Multiple Document Interface) Window. It is an assembly structure of eight child-windows, shown as Figure 4.3. There are one-to-one relationships between the main interface window and the child-windows, except for the Data Descriptor, that has a one-to-many relationship. A one-to-one relationship means that there is only one instance of a child-window corresponding to the main MDI window when the interface is created, while a one-to-many relationship indicates that many instances of the same object may occur in the user interface. The user interface

design maintains as many one-to-one relationships as possible to keep the interface simple and clean. The DFD Editor, ERD Editor, and DFD Descriptor show the descriptions of the current DFD and ERD. These windows are updated every time a user chooses to change to other diagrams. The multiple instances Data Descriptor on the user interface allows a user to cross-reference the definition of a data entity from top level to lower levels. Unlike other child-windows, the Data Descriptor is small and simple.

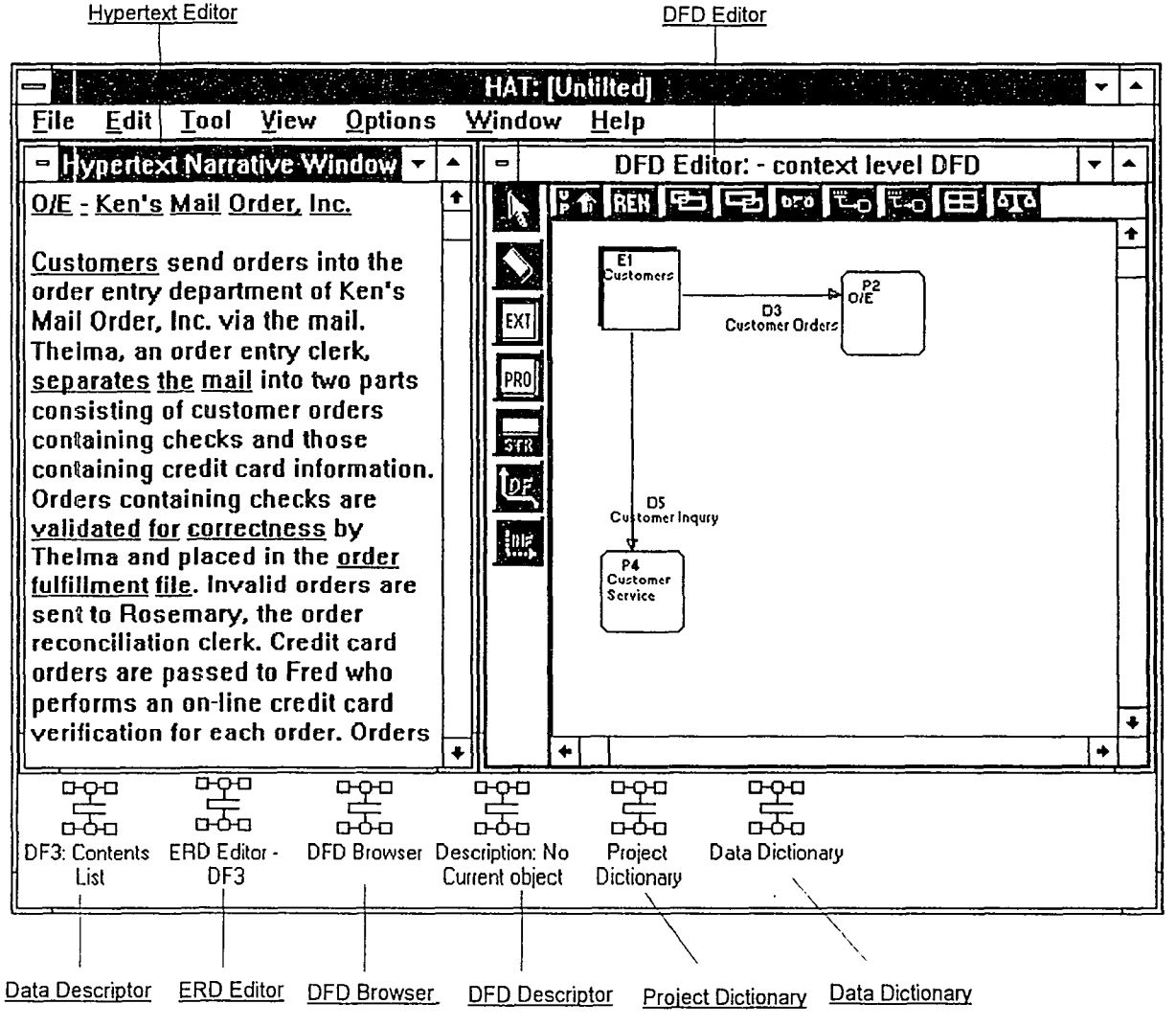


Figure 4.4. A snapshot of the user interface - default settings

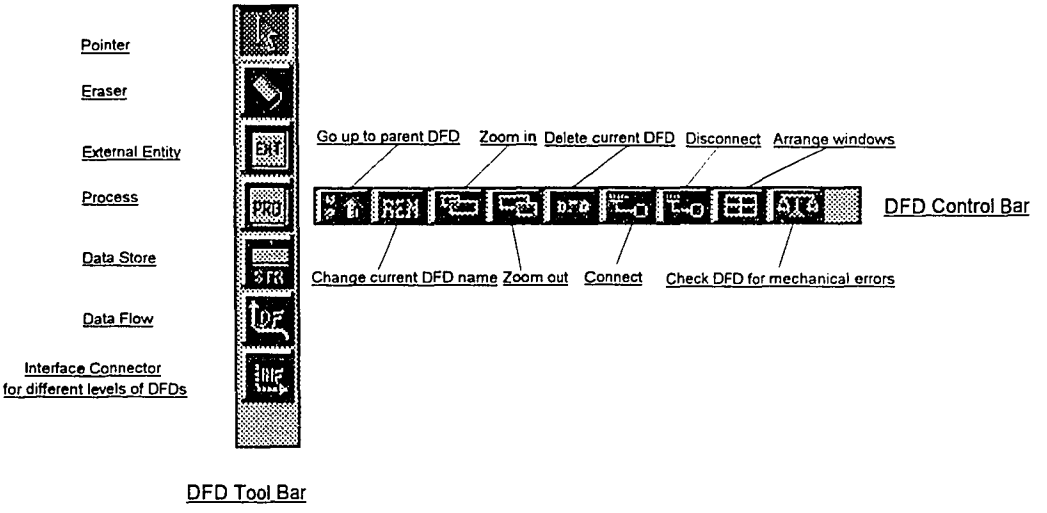


Figure 4.5. The Tool Bar and Control Bar of the DFD Editor

Figure 4.4 and Figure 4.5 shows the default setting of the user interface. The system assumes that a user starts with a problem description and builds a DFD model. Other child-windows are iconic to leave more space for the Hypertext Editor and DFD Editor. A user has the choice of re-configuring the user interface to start with ERD modeling instead of DFD modeling. The goal of this design is to keep the interface look and feel simple. Child-windows and other information are withheld until requested by users. Thus, the user can customize the work environment to suit individual preferences. Also, the system provides a one-touch button that sets all windows to default settings.

As one of the most important features of HAT, inter-connectivity of child-windows through hyperlinks is fundamental to the user interface. Figure 4.6 shows the channels for hyperlinks built in by the system. The *DFD Browser*, *DFD Descriptor*, *DFD Editor* and *Hypertext Editor* are connected with 'hot' hyperlinks (thick lines in Figure 4.5), which means that a change in any one of the windows will immediately update other windows. Connections among other windows are 'warm' or 'cold', which means no update until the window is activated or users choose to refresh the window.

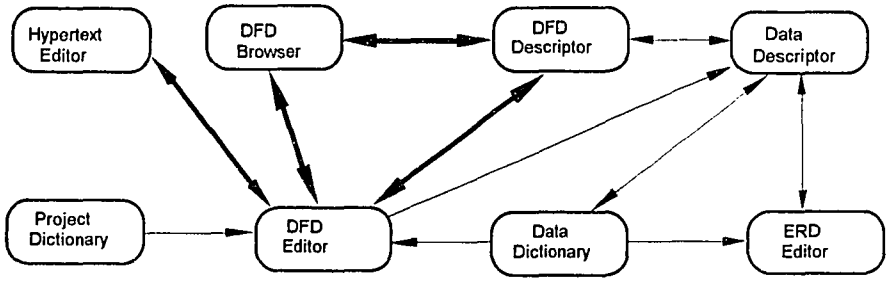


Figure 4.6. Channels for hyperlinks among child-windows

A 'hot' hyperlink is more intuitive and easy to follow for users, but it is also more difficult for the system to maintain. On the other hand, a 'cold' hyperlink consumes less system resources and is sufficient for less frequently used windows. The trade-off of 'hot' and 'cold' hyperlinks is determined by the considerations of interface functionality and implementation complexity.

The following sections are more detailed descriptions of the user interface. These descriptions reveal the fundamental classes and their structures that implement the hypertext-based user interface.

4.2.1 The Hypertext Editor

The *Hypertext Editor* is a hypertext window that allows users to specify problem narratives, create hyperwords and connect hyperwords to the graphical objects in the drawing windows. A typical early stage systems development scenario involves a gradual development of system specifications. The process is investigative and not linear. During an investigation, facts are uncovered gradually and each new fact leads to new questions that, in turn, lead to more facts. Therefore, a network of processes is constructed along with the investigation. Within this network, a process only makes sense in relation to other processes and specifications. Obviously, hypertext is an ideal medium to support this mode of work, in which hyperlinks created by users bridge one-dimensional text and

two-dimensional graphical models (DFD and ERD models). This 'hyper-dimension' is a major feature that this research adds to the approach of conventional CASE tools.

The Hypertext Editor bases its components on the classes of the Borland OWL (Object Windows Library). As indicated in Figure 4.7, the Hypertext Editor is inherited from *TFileWindow* of OWL and has two components: *HEditor* and *HyperwordList*. *TFileWindow* allows the Hypertext Editor to have basic ASCII file operations, that is 'New', 'Open', 'Save', 'Save As' and so forth. The Hypertext Editor redefines these operations for its special hypertext and hyperlink features.

HEditor inherits all the editing functions from *TEditor* of OWL, such as 'Insert', 'Delete', 'Cut' and 'Paste'. In addition to these basic editing functions, HEditor can highlight hyperwords with colors and underlining. Special care must be taken to create, display, delete, scroll, paint and repaint hyperwords, since *TEditor* does not provide any of these services.

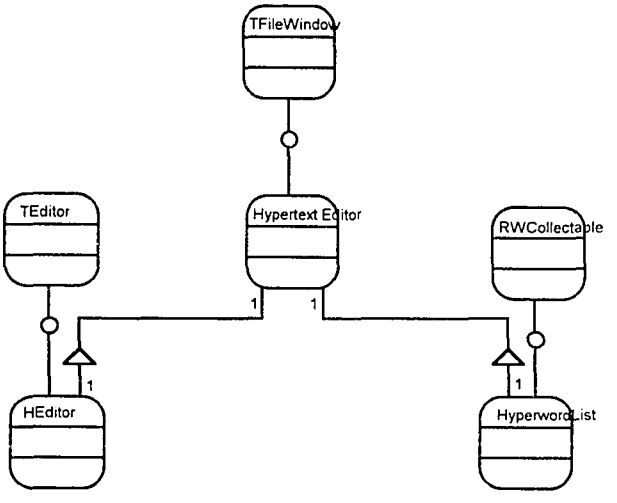


Figure 4.7. The Structure of the Hypertext Editor

HyperwordList is a local data manager that keeps track of all the hyperwords. Since the hypertext editing functions are added on top of standard *TEditor*, the

information about the positions and links of the hyperwords must be kept in temporary storage. This information is used by the HEditor to paint and position the hyperwords. HyperwordList also keeps the information on hyperlinks from a hyperword to a DFD or an ERD graphical object. It holds the key to bridge the narrative text and graphical models. The hypertext and hyperlink information is stored and retrieved persistently as part of the project data file. The basic saving and retrieving functions inherited from *TFileWindow* are over-written by persistent saving and retrieval methods that are inherited from the *RWCollectable* object of Tool.h++. Since the central data repository is also built on the basis of Tool.h++ classes, HyperwordList is fully compatible and cooperative with the data repository.

4.2.2 The DFD Editor and the ERD Editor

The DFD Editor and ERD Editor are the tools for users to create graphical models. Although functionally and logically DFDs and ERDs are quite different, the DFD Editor and ERD Editor share similar class structures and inherit from the same graphical library as shown in Figure 4.8. All the graphical-relevant windows are derived from *GWindow* of ObjectGraphics of Whitewater Group. Both the DFD Editor and the ERD Editor are derived from *ObjWin*, which contains the common features used by both editors to create and maintain graphical objects, such as moving, adding, zooming and deleting. Both the DFD Editor and the ERD Editor use pre-defined icons to draw the diagram. They have similar components: a drawing canvas for graphical objects, a tool-bar palette for pre-defined tool, and a control-bar palette for special operations. General features of these components are represented in the objects of *DrawWind*, *ToolBar*, and *ControlBar*, from which DFD and ERD-specific canvas, tool-bar and control-bar are derived. A user can choose these pre-defined functions to point, select, drag, and draw graphical objects.

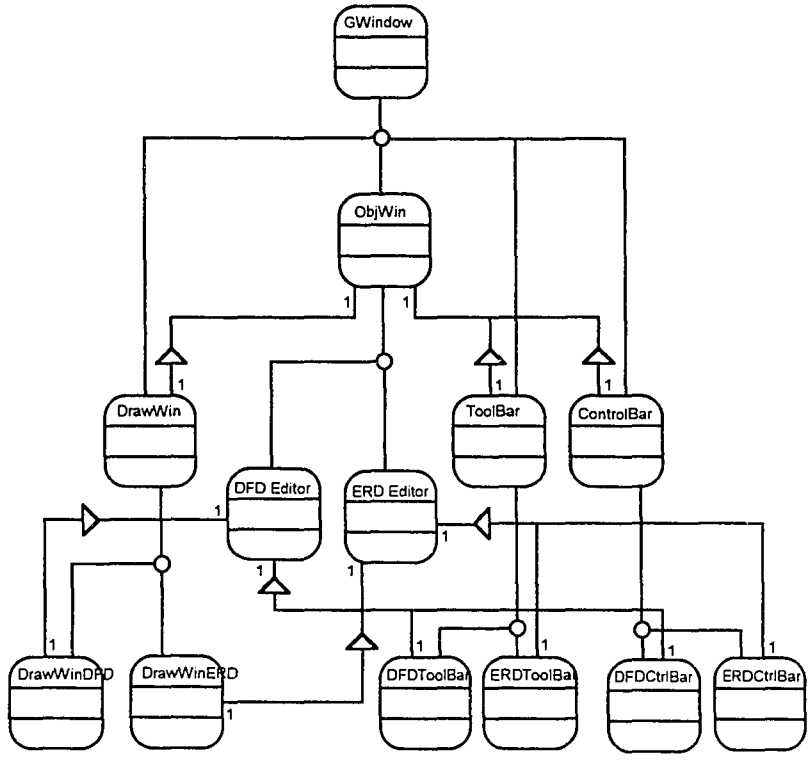


Figure 4.8. The Structure of the DFD Editor and ERD Editor

The basic symbols in *DFDToolBar* palette (*data flow, data store, process* and *external entity*) follow the Gane and Sarson method [Gane 79]. The *DFDCtrlBar* palette contains a set of control buttons for different operations including navigating from one level of DFD to another level, connecting and disconnecting to a hyperword, renaming or deleting a current DFD, and triggering a floating menu to pop-up the DFD checking rules. The *DFDDraw* differs from the ERD drawing canvas in that it connects to the DFD-related data structures in the data repository and updates entries to the project dictionary. DFDDraw also has the 'hot hyperlinks' that connects to DFD Browser, DFD Descriptor and Hypertext Editor to update changes in the graphical window.

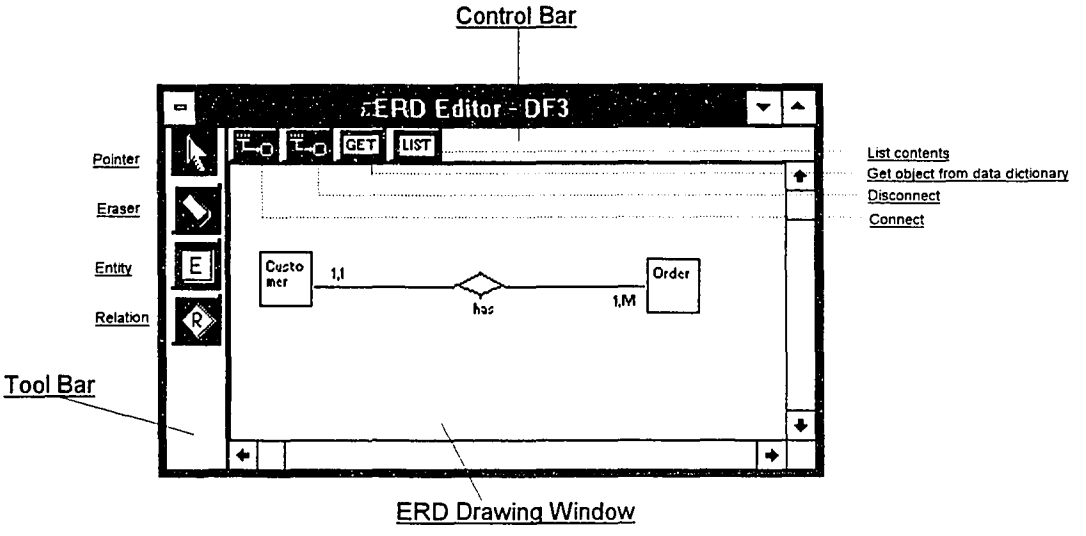


Figure 4.9. A snapshot of ERD Editor

The *ERDToolBar* contains the symbols necessary to draw ERD models using a variation of the Chen method [Chen 76]. The control bar of the ERD Editor is much simpler than that of the DFD Editor as seen in Figure 4.9, because an ERD does not require a hierarchical structure to represent different levels of models. However, buttons to create and delete hyperlinks are necessary. Similar to the DFD Editor, the ERD Editor has a direct connection to the data analysis section of the data repository. It also has a 'warm hyperlink' to Data Descriptor so that an entry to either Data Descriptor or ERD Editor will be displayed in both windows. In addition to the hyperlink button, two more buttons are used to set up links to the Data Dictionary and Data Descriptor windows. The 'GET' button will 'grab' an existing data record in the data dictionary into the current ERD. The 'LIST' button brings up the Data Descriptor corresponding to the current ERD.

There can be two kinds of ERDs: *Conceptual ERD* and *Implementation ERD*. A Conceptual ERD describes the overall aspects of data relationships. It is used as a

starting point for data analysis and may contain data entities that may not exist in an implementation model. An Implementation ERD is used to describe the data relationships for implementation purposes. It contains more information and reflects data entities that will be implemented as files or relations. The ERD Editor supports both conceptual and implementation ERDs and links them to the Data Descriptors and the Data Dictionary.

4.2.3 The windows for process analysis

Process analysis is a description of the flow of data from one process to another. In HAT, process analysis is described using layers of DFDs, starting from the context level to a more detailed level of description. The DFD Editor is a visual graphical tool that a user uses to interactively draw DFD models. In addition to the graphical model, other tools are needed to navigate from one DFD to another, manage the descriptive text of each graphical object, and keep track of the project dictionary. Figure 4.10 shows snapshots of the three windows that perform this task: DFD Browser, DFD Descriptor, and Project Dictionary.

In a hypertext system, a browser, a map or other navigation tools are often provided to guide users through the information web. Without proper orientation tools, users may easily get lost and cannot perform effective information retrieval. The problem of 'getting lost' has been recognized in a lot of literature [Conklin 87]. HAT incorporates a DFD browser to outline the layers of DFDs in a process analysis. The *DFD Browser* is a window that contains a list-box. Entries to the list-box are names of DFDs and graphical objects in a DFD. A double-click on a graphical object entry will bring up the graphical object in DFD Editor; a double-click on a DFD name will toggle to open or close the list of all the objects in the DFD.

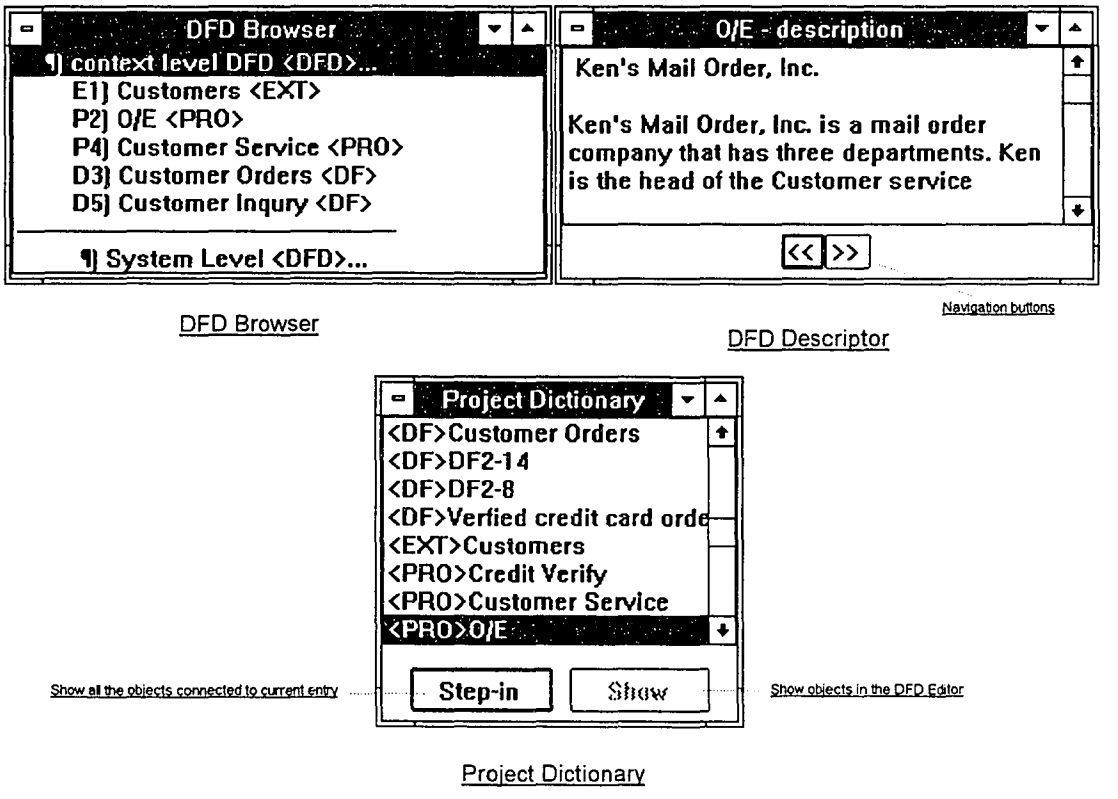


Figure 4.10. The windows for process analysis

DFD Descriptor is a HyperCard-styled tool to manage the description of each graphical object in a DFD. The text in the Hypertext Editor is often used to describe a problem in general. The hyperwords in the hypertext window are connected to different graphical objects depending on the user's choice. The DFD Descriptor, on the other hand, contains a short description dedicated to a specific graphical object. This description can be a detailed description of the object in addition to the general description in the hypertext window. The two buttons at the bottom of DFD Descriptor are used to browse through the graphical object in current DFD.

The Project Dictionary window shows the contents of the project dictionary. It provides a tool for users to directly observe what is in the project dictionary. In addition, the two buttons ('Step in' and 'Show') are used to list all the DFD graphical objects

associated with a selected dictionary entry and to show the object in the DFD Editor, which provides another way to retrieve graphical objects.

4.2.4 The windows for data analysis

Data analysis focuses on describing the data entities and relationships among them. An ERD is a tool used by many system analysts to describe data models. HAT includes a data dictionary in its data repository that contains all the data-related objects, such as data flows, data stores, data records, and data elements. For each data-related object, except for a data element, there can be a list 'exploded' to describe its contents and attributes of the object. The Data Descriptor is used to display and maintain these contents lists. As data analysis proceeds, a network of data-related objects evolves. This network is called the Data Relation Graph in HAT (see Section 4.3 for detailed descriptions).

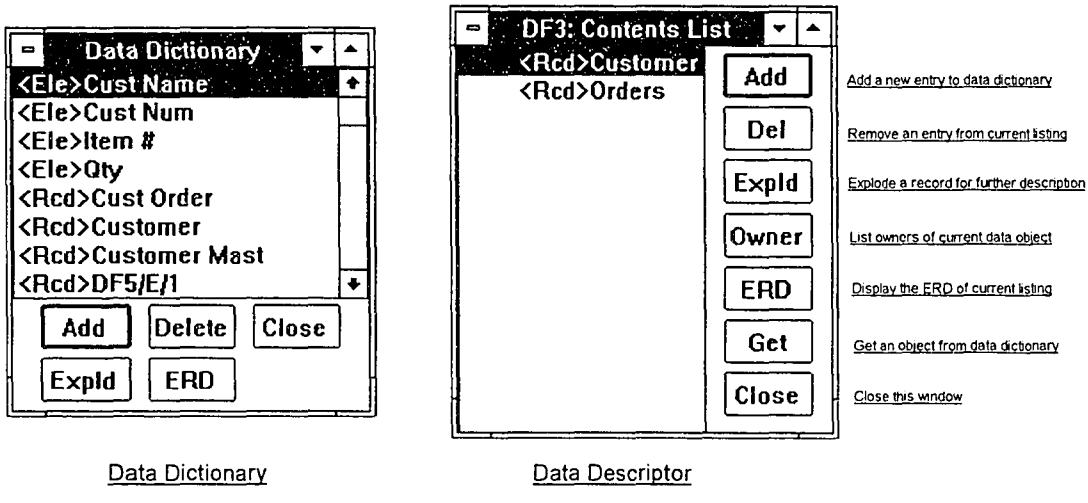


Figure 4.11. The windows for data analysis

As indicated in Figure 4.11, Data Dictionary and Data Descriptor are lists of data-related objects. On one hand, the data dictionary is a non-duplicate list that contains

references to all data objects regardless of their relationships. On the other hand, an instance Data Descriptor represents the 'parents-children' relationships in the Data Relation Graph. The 'OWNER' and 'EXPLD' buttons in a Data Descriptor window can display different levels of data objects in the Data Relation Graph. Both Data Dictionary and Data Descriptor windows have direct connection with the ERD editor that will display either a conceptual ERD to the whole project or an implementation ERD specific to the selected data object.

4.2.5 Comments on the user interface design

The HAT user interface focuses on the implementation of hyperlinks through the hyperlink channels described in Figure 4.5. These hyperlinks allow users to retrieve the same information in many different ways. At the same time, to fulfill the prescribed functions, the user interface employs one-touch buttons to trigger most of the operations, so that users can visually touch and feel the effect of the changes. Several easy-to-use dialogue boxes are also used for DFD and ERD inputs as well as simulation parameter definitions (see Section 4.5.4 for more details). The WIMP (Window, Icon, Menu and Point) devices are inherited from standard Windows classes that have uniform format and are familiar to users.

The commands and buttons used in the user interface are consistent among different child windows. Some of the buttons are decorated with bitmaps for easy recognition. Since the system is developed in C++, the system response time is good. Common DFD and ERD modeling errors are handled by the system. Although much effort has been made to improve the user friendliness of the user interface, some of Galitz's user interface design criteria [Galitz 93] (see Section 3.1.2) are not met. The

most obvious flaw is error recovery. The user interface has not included an 'undo' function in the current design.

The author observed the beta testing of HAT among a group of MIS undergraduate students and found that most of the students could learn the basic functions within one class session (less than an hour). Because most of the students had had experience with Windows applications, they were not intimidated by the appearance of the HAT user interface. On the contrary, some students became very involved in testing and debugging this new tool and gave a lot of good suggestions for improvement. Behavioral studies need to be done before the effectiveness of HAT can be measured. The author's expectation is to see a combination of the 'width' of user acceptance as a result of hypertext techniques and the 'depth' of user understanding because of dynamic evaluation with simulation techniques.

4.3 The Data Repository Subsystem

The data repository subsystem is the central data storage for HAT. A centralized data repository is the nucleus for interaction and cross-reference of the multiple-window user interface. In an object-oriented system, the data repository operates on an object basis; an object, instead of a data element, is the basic transaction unit.

Since HAT is a front-end CASE tool that has no systems design functionality, the data repository is much simpler than would be found in the typical large CASE tool. As indicated in Figure 4.12, the data repository subsystem is composed of four components: the project dictionary, the DFD tree, the data dictionary, and the data relation graph. Currently, HAT is limited to a file-based system. The information of a project is stored in a single file, which is not sharable concurrently. A third-party software package -

Tools.h++, is used to manage object-oriented persistent file operations. Thus, the design of the data repository focuses primarily on the management of objects in memory.

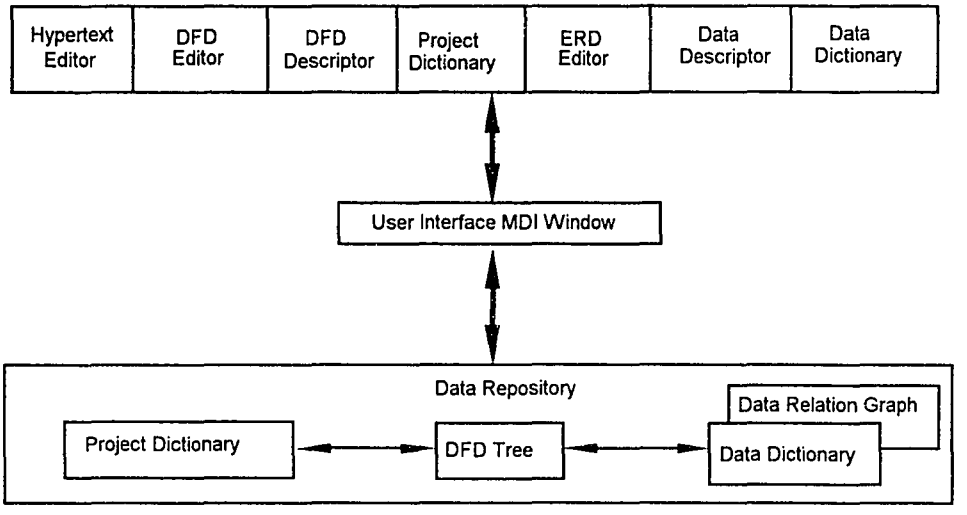


Figure 4.12. Connections between the user interface and data repository

The project dictionary itself contains a set of *project entries* that keep track of conceptual information about a DFD object, such as names, types and descriptions. Other DFD-related objects keep pointer references to the dictionary entry. Duplicate entries are not allowed. A link-list is used for each *project entry* to keep track of DFD objects that appear more than once in the project. This also serves as a back-pointer to the DFD tree.

The data dictionary manages the descriptions of the data items in a *data flow* or a *data store*. Similar to the project dictionary, the data dictionary is composed of a set of *data entries*. A *data entry* contains a *data object* and its connection lists. As analysis progresses, a network that describes the ownership of one data object by another will evolve. This network of *data objects* is called the '*data relational graph*', from which ERD diagrams can be constructed and maintained.

A tree structure is used to represent the step by step decomposition process of systems analysis. The top level DFD is the root of the DFD Tree. All other DFDs are generated by a step-wise refinement as the children or grandchildren of the top level DFD. Inside a *DFD Tree*, each DFD contains a number of *DFD Objects*.

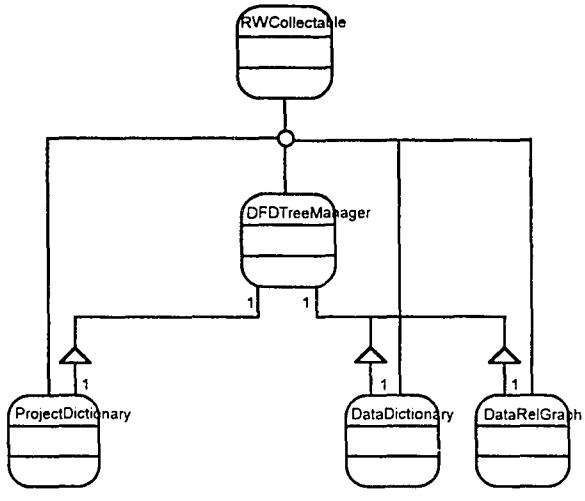


Figure 4.13. Structure of the data repository

Since the data repository is built on top of Tools.h++, all the objects in the data repository are derived from *RWCollectable* of Tools.h++ (see Figure 4.13). An assembly structure of *DFDTreeManager* assembles the objects of the project and data dictionary as well as the data relation graph. The DFDTreeManager also keeps a pointer to the root and a pointer to the current DFD in the DFD tree so that it can save and retrieve the whole DFD family, or set any of the DFDs to current.

4.3.1 The DFD tree

Systems analysis is a step-wise refinement process from the outlines of a problem to more detailed problem descriptions. This process naturally generates layers of documents with a hierarchical structure. When a DFD is used for systems analysis, a DFD tree similar to Figure 4.14 will be generated through the analysis process. In this

example, the *Root* pointer of DFDTreeManager constantly points to the root of the DFD tree - the Context Level DFD 0, and the *Current DFD* pointer floats around to point to the DFD currently displayed (in this example, the Detailed Level DFD 1.1).

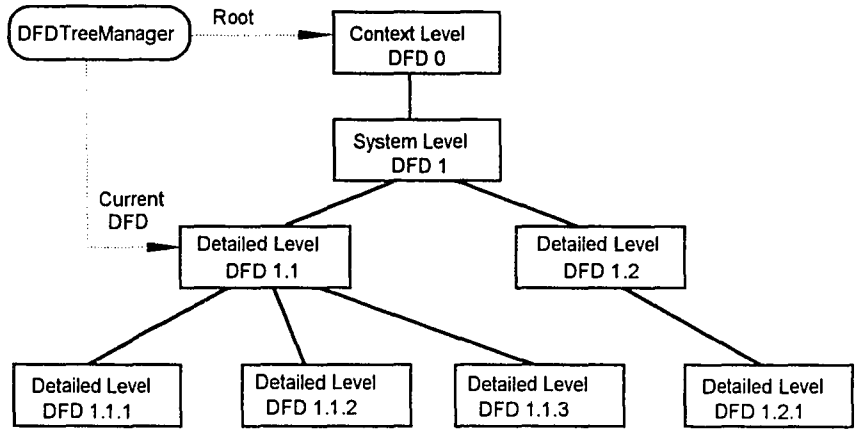


Figure 4.14. An example of a DFD tree

The object that describes a DFD has an assembly structure of other objects: *ConceptualDFD*, *VisualDFD*, *ChildList*, *ParentList*, *NodeList*, *FlowList*, and *SimRun*. As described in Figure 4.15, all the objects have the same ancestor, the *RWCollectable* of Tools.h++. Most of the objects of the data repository subsystem are descendants of RWCollectable to take advantage of the object-oriented persistent storage and retrieval feature in the object.

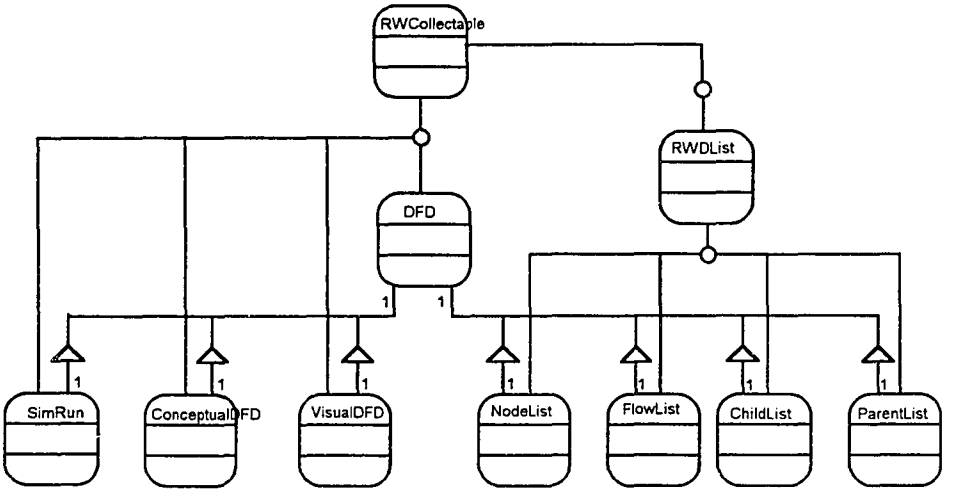


Figure 4.15. Class structure of a DFD

A ConceptualDFD represents the conceptual features of a DFD, such as the DFD name and special labels. The VisualDFD captures the visual aspects of a DFD that describe how a DFD should be displayed in the DFD editor. Four lists are used to keep information of DFD nodes and DFD flows, as well as the pointers to the lower and higher level DFDs in a DFD tree. In most of the cases, a parent list contains only one pointer to the parent DFD. All the lists inherit from the *RWDList* class, which is a standard double linked list provided by Tools.h++. RWDList itself is a descendent of RWCollectable and has persistent storage and retrieval capacity. A SimRun object keeps the information necessary to carry out a simulation experiment (number of runs, warmup period, and run length).

The content of a NodeList is a cluster of *DFDNodes* that are derived from *DFDObject*. All DFD symbols with node features are descendants of DFDNode. They are *ExternalEntity*, *Process*, *DataStore* and *InterfaceNode* (see Figure 4.16).

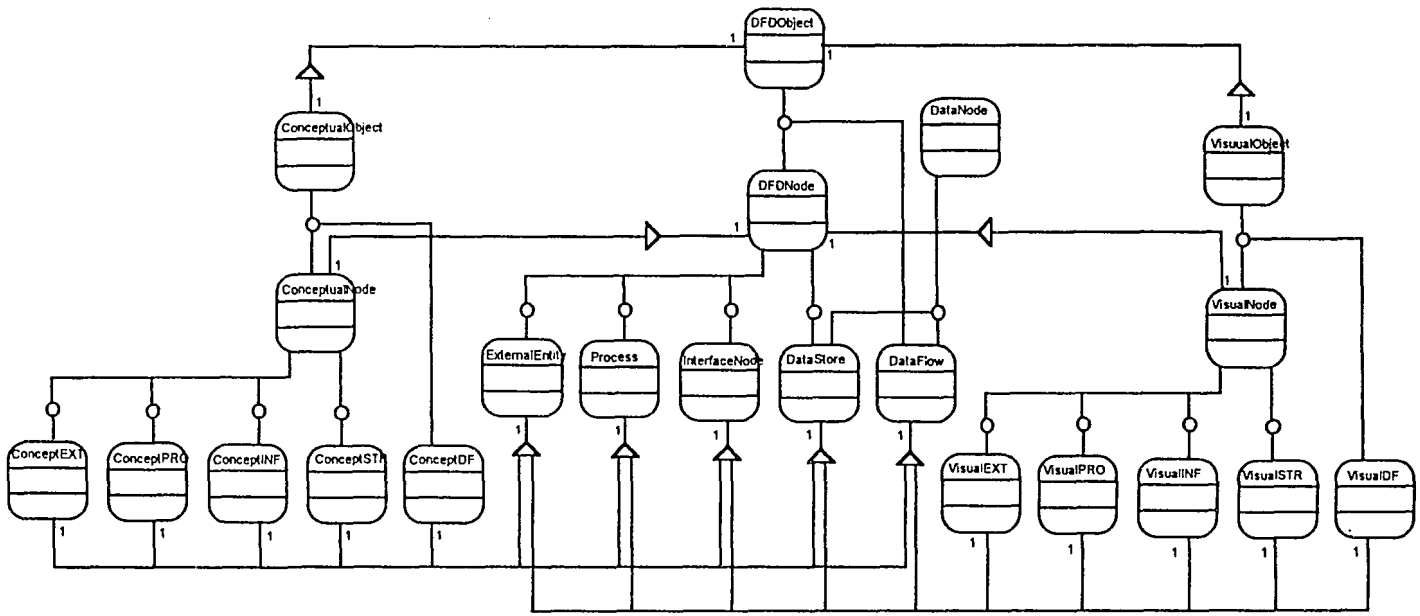


Figure 4.16. Structure of DFD objects

Figure 4.16 also indicates that a *DataFlow* object is another descendent of DFDObject. A DataFlow has different features than the nodes. Nevertheless, it inherits common features of a DFDObject. The FlowList in a DFD is composed of a set of DataFlow objects. It is also worth noting that both DataFlow and DataStore inherit features from *DataNode.* This multiple inheritance enables a DataFlow object or a DataStore object to acquire the features of both DataObject and DFDObject.

Systems analysis requires that a process node in a DFD has the ability to explode to a detailed DFD to describe further analysis of this process. Therefore a *Process* object has a special pointer pointing to its next level DFD. A reference to a next level DFD pointer is also added to the child list of the DFD that contains the parent process. An *ExternalEntiy* object is a mirror image of an external entity in a DFD that represents a source or destination of a data flow. An *Interface* node is a special node defined to connect data flows from one level of DFD to another.

Similar to the structure of a DFD, the structure of a DFDObject has three sections: the left section, the central section, and the right section. A cluster of conceptual objects constructs an object sub-tree on the left side of Figure 4.16 and a cluster of visual objects on the right side of the diagram. Different levels of conceptual and visual objects are connected to their DFD-related objects via assembly structure linked to the central part of Figure 4.16. The separation of conceptual and visual aspects of a DFD-related objects makes it easier to modify and maintain.

A further description of the structure in Figure 4.16 is given in Figure 4.17. The conceptual aspect of a DFD-related object holds a direct connection to a project entry corresponding to this object in the project dictionary, from which the information of name, label, type and description of the DFD object is available. There is a one-to-many

relationship between a ProjectEntry and a DFDObject. Because multiple DFDObjects may have the same name, type, and label, they have the same ProjectEntry. On the other hand, the visual aspects of a DFD-related object contain the physical location information in the DFD editor. The central part of a DFD-related object bridges the conceptual and visual aspects. A DataNode object maintains lists of DataFlows coming in and out of it. A Dataflow object contains the DataNode pair on the ends of the flow. The connection information is accessible from both conceptual and visual sides.

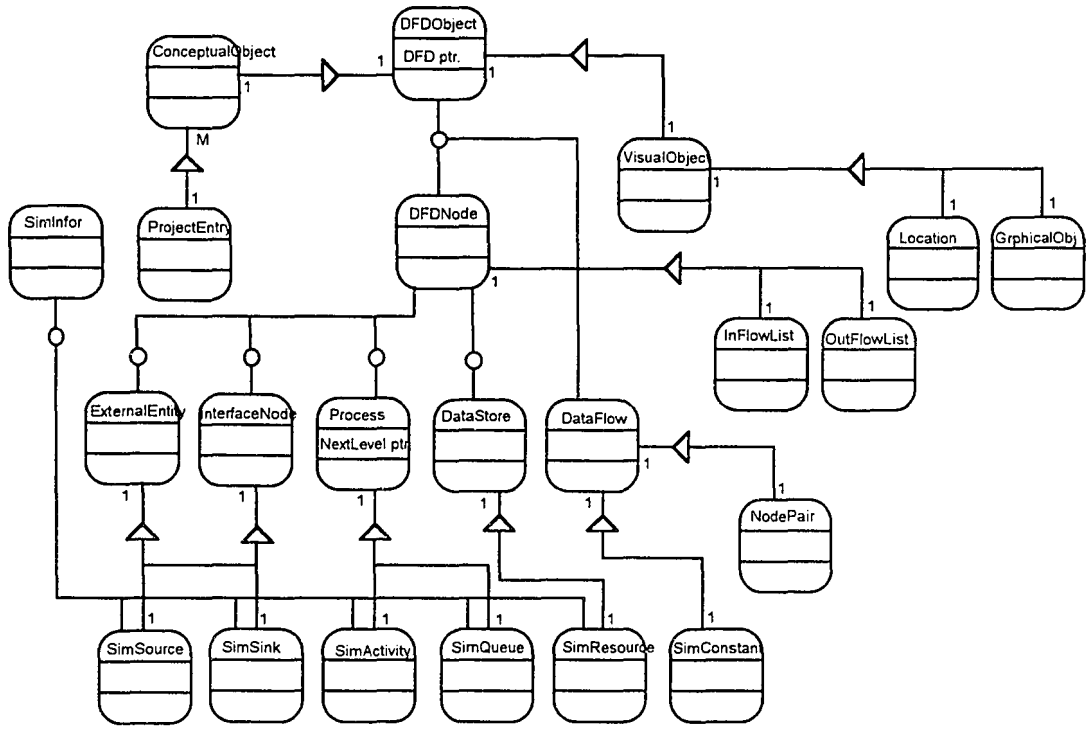


Figure 4.17. A further description of DFD-related objects based on Figure 4.16

In addition to the objects necessary for DFD drawing, simulation information is also stored in the data repository. Each DFD-related object has at least a descendant of *SimInfor* object attached to it that contains information necessary to carry out simulation modeling. Simulation results are also extracted and stored in SimInfor objects. The arrangement for SimInfor objects is shown at the bottom of Figure 4.17: an external

entity or an interface node is either a source or a sink in a simulation model; a process corresponds to a pair of a queues and an activity in a simulation model; a data store is viewed as a resource used by a processes; and the time used for a data flow operation is assumed to be constant.

4.3.2 The data relation graph and structures for the ERD

The data relational graph of a project is a list of *DataNode* objects. Each DataNode is the data-related section of a DataFlow or a DataStore. Further descriptions of a DataNode by a set of DataRecords and DataElements becomes a tree of data descriptions (data tree). Multiple DataNodes construct a forest of data trees. A DataRelationGraph object holds the root of each data tree and provides services to expand, trim and maintain the forest.

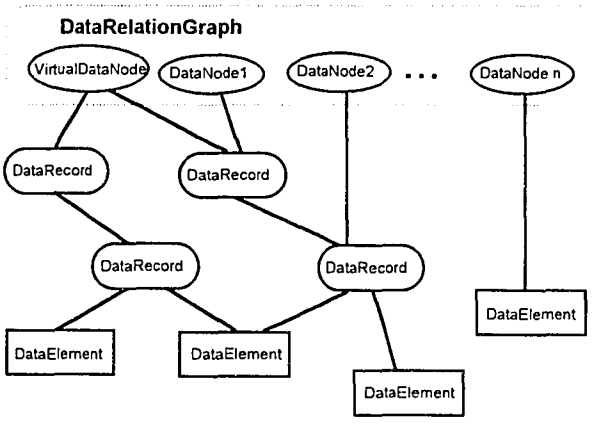


Figure 4.18. An example of a data relation graph

Figure 4.18 illustrates an example of a data relation graph. When a data flow or a data store is created, a DataNode object will be created. The DataNode is added as a root to a data tree to the DataRelationGraph. Each DataNode has the potential to have children as the analysis progresses and more details of data descriptions are added. These children can be DataRecords and DataElements. A DataRecord has the potential to be

further described by other DataRecords and DataElements. However, a DataElement is a terminator of a data tree that represents a basic component of a data description. A DataRecord or a DataElement may belong to a different parent data objects or share the same parents with others. The responsibility of a DataRelationGraph is to manage such a network of data objects and provide services to other parts of the system. A '*VirtualDataNode*' is added to a DataRelationGraph to hold the data tree of a conceptual ERD for data analysis. The VisualDataNode has no connection with any DFD object, so that data analysis can be carried out independent of process analysis.

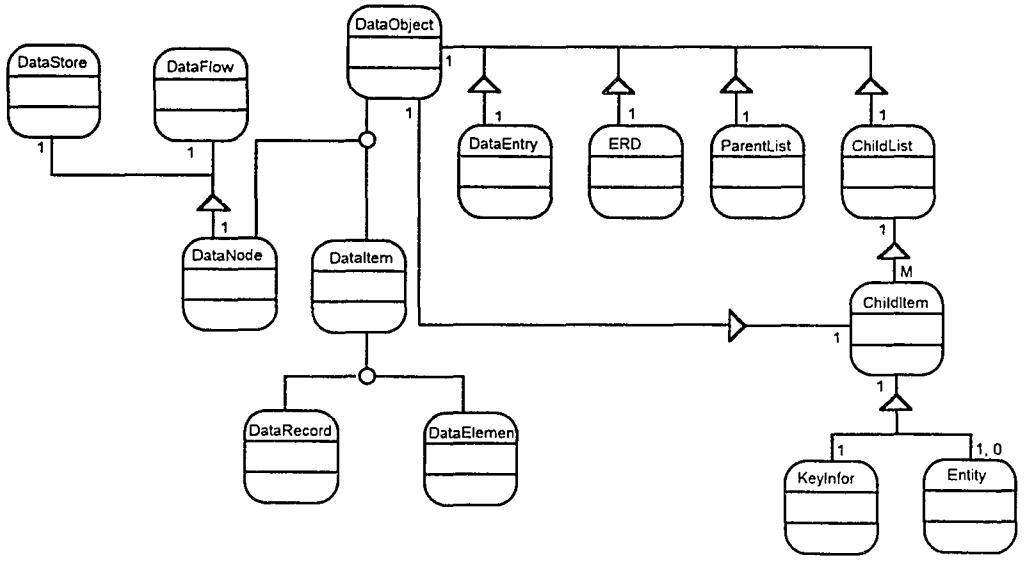


Figure 4.19. The structure of data-related objects

A more elaborate description of the structure of a data object is presented in Figure 4.19. The diagram shows that DataNode, DataRecord, and DataElement inherit *DataEntry*, *ERD*, *ParentList*, and *ChildList* from *DataObject*. A DataNode always has either a DataFlow or a DataStore connection in a DFD tree. That is where the DFD and the ERD are connected. A DataItem object represents those data objects that are not directly associated with a DFD object and this group is further decomposed into DataRecords and DataElements. Since a DataElement cannot have any child and ERD

connection, special methods have to be provided to invalidate operations on ChildList and ERD.

A Childlist contains a list of ChildItems. Each ChildItem contains a DataObject that is listed as a child to the parent. The KeyInfor object in a ChildItem represents the relative position of the child with respect to other children in the list, i.e. if a child is a primary key, a secondary key or a non-key member in the child list. An Entity object may also be attached to a ChildItem, whose DataObject member is either a DataNode or a DataRecord, to represent an entity in the ERD attached to the parent data object. For a ChildItem whose DataObject member is only a DataElement, there is no Entity object attached to it, because it will not show up in any ERD.

Because multiple occurrences of a data object can be represented by connections to multiple child lists, it is not necessary to duplicate a data object in a data relation graph. Though it may appear as a member of more than one ChildItem, a data object has a one-to-one relationship to a data entry in the data dictionary. An ERD connection only occurs in a DataNode or a DataReocrd. An ERD is used to describe the relationship of the ChildItems in the child list. Therefore, a DataObject can at most have one ERD associated with it.

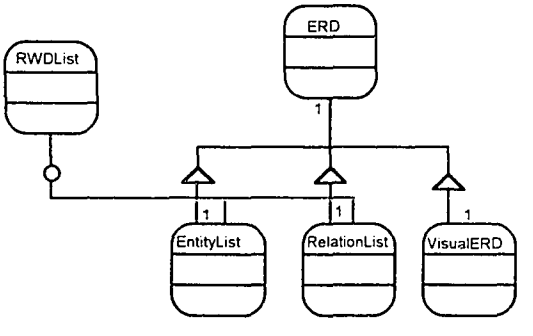


Figure 4.20. The structure of an ERD

Figure 4.20 shows the structure of an ERD object. It contains an EntityList and a RelationList. Both of the lists are inherited from RWDList to keep object persistence. Similar to DFD structure, a VisualERD is used to represent visual aspect of an ERD. An ERD object manages its component objects and provides insertion, deletion and retrieval services to the ERD editor of the user interface.

The contents of the two lists in an ERD are described in Figure 4.21. An Entity object has a pointer to the DataObject associated with it. Therefore, all the information of the data object is shared by the entity. In addition to the connection to its data object, an Entity object has a RelationList to keep records of all the relationships connected with it. Physical location and connection to the graphical object shown in the drawing window are also included in an Entity object.

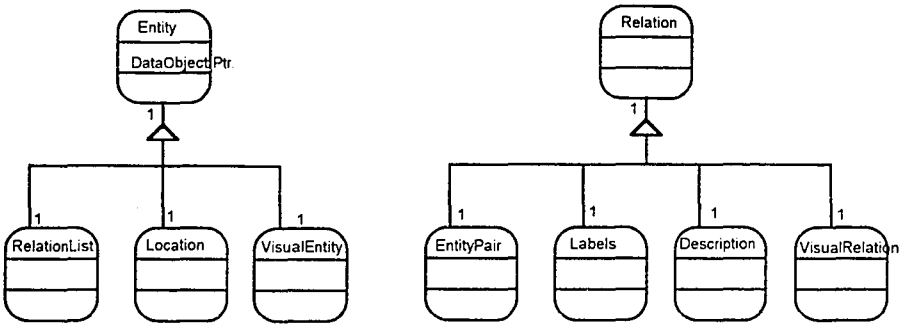


Figure 4.21. The structure of an Entity and a Relation

Since a relation is only significant in the context of entities connected by it, a Relation object contains an EntityPair that represents the two entities at both ends of a relation. A relation is not listed in the data dictionary. Descriptions of a relation are kept locally inside a Relation object. As in the previous cases, a visual object has to be included to represent the graphical object on the ERD drawing canvas.

4.3.3 The dictionaries and their entries

Both ProjectDictionary and DataDictionary are collections of ProjectEntries and DataEntries. There is no duplication in the dictionaries. Figure 4.22 shows that the two dictionaries are derived from RWSet class of Tools.h++. The nature of set operations guarantees the uniqueness of each entry in the dictionaries. Both of the dictionaries have a back-pointer to the DFDTreeManager so that they can easily access information in the DFD tree. Basic services provided by the dictionaries are similar, but differ in their entries to the dictionaries.

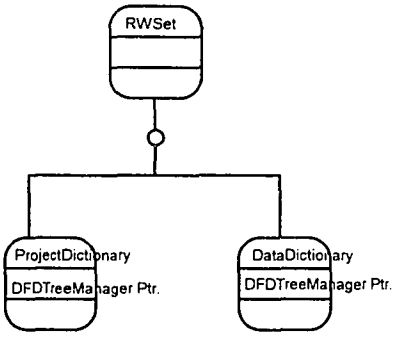


Figure 4.22. The origin of a project dictionary and a data dictionary

A ProjectEntry and a DataEntry are different in the way that they hold information for different models. A ProjectEntry holds information of a DFD-related object, while a DataEntry holds that of a data-related object. Since a ProjectEntry may have one-to-many relationships with DFD-related objects, a ConnectionList is used to keep track of all the DFDObjects that are associated with the project entry (see Figure 4.23). A DataEntry has only a one-to-one relationship with a DataObject. Therefore, a DataObject pointer is sufficient for a DataEntry to find its related data object. Both ProjectEntry and DataEntry have *Labels* and *Description* objects to store the information on the label, type, name, and free-format description of the corresponding DFD or data object.

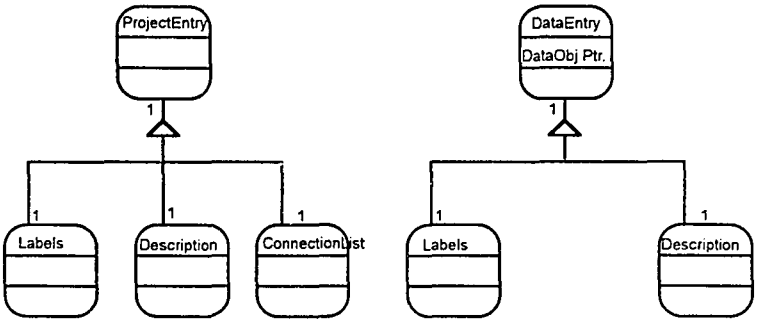


Figure 4.23. The structure of ProjectEntry and DataEntry

The data repository subsystem manages objects of various kinds with carefully designed structures and provides services necessary for the user interface to create system models visually and interactively. The data repository also provides services to the simulation subsystem for simulation modeling and storage of simulation results.

4.4 The DDE Interface

The keys that hold the HAT subsystems together are the DDE data links. Each subsystem depends on DDE links to transfer scripts of simulation modeling, questions and answers, as well as other control information.

A DDE link creates a client/server (destination/source) relation during the execution of two applications. The link will remain connected until one of the application requests disconnection or terminates execution. There are three basic modes for DDE data transfer as shown in Table 4.1. An application can be both a server and a client.

Table 4.1. Basic modes for DDE data transfer

Request	A client initiates a request to its server for certain data items and the server replies with the requested data.
Automatic	A server monitors the data buffer and automatically updates changes to its clients.
Poke	A client sends a short message to its server to relay special notice.

4.4.1 The data interface structure

HAT is composed of three separate Windows applications. Each application should have the ability to 'talk' with the other part of the system fluently in a bi-directional fashion. It is obvious that a DDE data interface embedded in each of the applications should have the ability to serve as both a client and a server for data exchange so that data can be transferred in both directions.

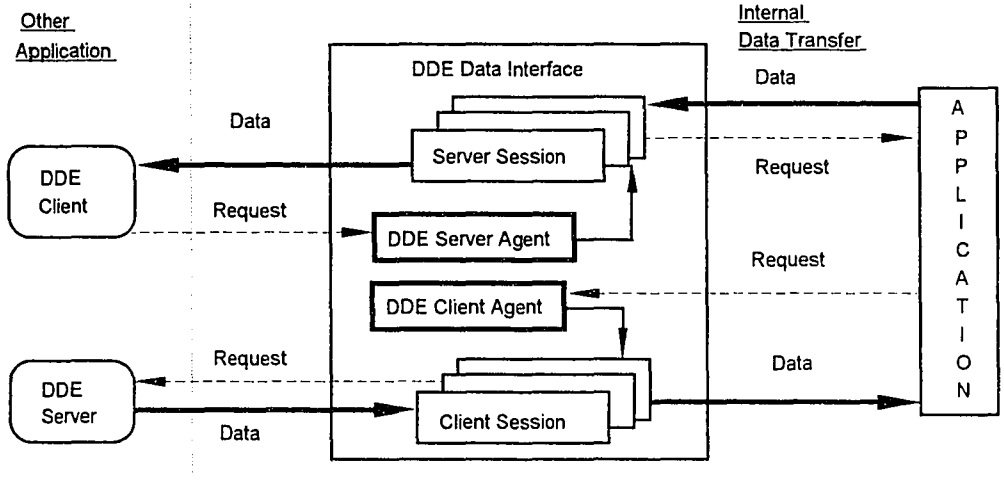


Figure 4.24. The DDE data interface in HAT

Figure 4.24 illustrates the concept of the DDE data interface used in HAT. Each DDE data interface contains a client agent and a server agent. An internal data channel is created between a DDE data interface and the application associated with it through message passing and function calls. The client agent manages all the client sessions created during the execution. When the application has a data request, it forwards the request to its DDE client agent. The DDE client agent creates a client session, sets up a DDE data link and requests the data from its DDE server. If the application requests an *automatic* data link with a server, the client agent chooses the automatic mode and forwards new data to the application whenever it is available. The client session can also *poke* short messages to its server upon request from the application.

The server agent handles data requests from other applications. For each new request, it sets up a server session. The server session then forwards the request to its application, where data is prepared according to the request. The server session returns the data to the client that requested the data. In automatic mode, the server session will monitor the changes in application through the internal data channels and start automatic data transfer whenever a change occurs.

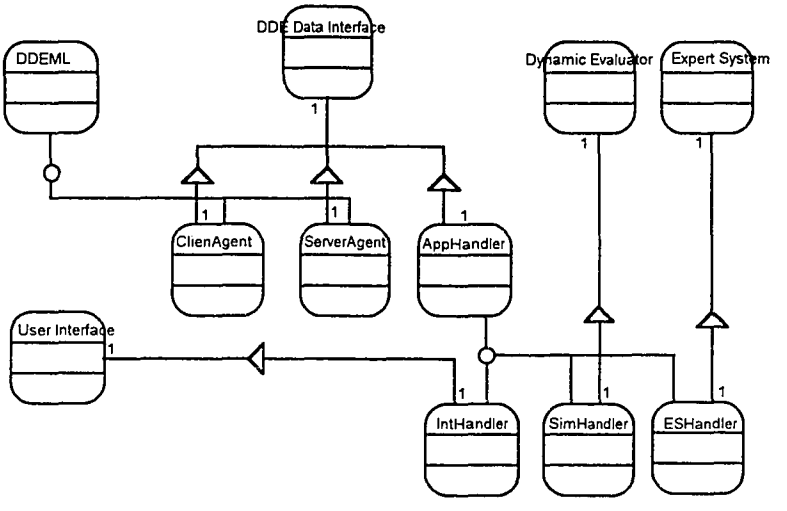


Figure 4.25. The class structure of DDE Data Interface

Figure 4.25 describes the objects that are used to implement the DDE data interface described in Figure 4.24. Microsoft provides a DDE Manager Library (DDEML) on top of Microsoft basic DDE utilities. The ClientAgent and ServerAgent in a DDE Data Interface inherit from DDEML to obtain the ability for basic DDE operations.

An AppHandler is the connection of the application attached to a DDE Data Interface, where the internal connections for data exchanges are constructed. Different subsystems have different data requirements. Therefore, they need different connections to a DDE data interface. The AppHandler and its descendants handle all the different

subsystems and make the implementation of ClientAgent and ServerAgent independent of the subsystems. A subsystem can take advantage of DDE data links once it has the right application handler[1].

4.4.2 The conversation protocols of the subsystems

The basic DDE 'hand shaking' protocols allow different applications to setup DDE links for data transfer. However, to make a data transfer process meaningful, a set of application-specific protocols is needed to define the conversations among applications.

There are three bi-directional DDE data links in HAT: the user interface to/from the simulation subsystem; the user interface to/from the simulation expert system; and the simulation subsystem to/from the simulation expert system. The DDE conversations are conducted as following:

1. The *user interface* to the *simulation subsystem:* A user creates a DFD model from the user interface and requests dynamic analysis. The user interface will convert the DFD model into a model script and forward the script to the simulation subsystem. The procedure is:

    i) The user interface 'locks' the DFD model and puts the model script in the DDE transfer buffer

---

[1] The simulation expert system is implemented in Visual Basic and parts of the DDE features are built-in. The implementation of DDE data interface and ESHandler is a little bit different from the user interface and the simulation subsystem. However, DDE data interfaces for all the three Windows applications in HAT are conceptually identical.

ii) The user interface starts a DDE data link to the simulation subsystem (the simulation subsystem as the server and the user interface as the client, Interface ⇨ Simulation), if the link has not been previously established. The simulation subsystem will be executed if it is not currently active.

iii) The user interface *pokes* a message '*Simulation model ready*'.

iv) Upon receiving this message from the user interface, the simulation subsystem initiates a DDE link to the user interface (the user interface as the server and the simulation subsystem as the client, Simulation ⇨ Interface), if the link has not been previously established.

v) The simulation subsystem *requests* for the simulation model script through the DDE data link.

2. The *simulation subsystem* to the *user interface*: After the simulation subsystem generates a simulation model based on the model script and runs the simulation, the simulation results are sent back to the user interface. If the simulation cannot complete because of incomplete model scripts or simulation run time errors, the simulation subsystem also reports the failure to the user interface.

   i) The simulation subsystem checks if the DDE data link to the user interface is still open. If not, the links are re-initiated.

   ii) The simulation subsystem *pokes* a message into the user interface '*Simulation finished*' or '*Simulation failed because of*.... (reasons)'.

   iii) Upon receiving a successful message from the simulation subsystem, in response, the user interface sends a *request* the simulation results and

distributes the results to the corresponding DFD objects in the data repository. If a failure message is received, the user interface will inform the user of the failure.

iv) The user interface 'unlocks' the simulated DFD model.

3. The conversation between the *user interface* and the *simulation expert system*: The simulation expert system is consulted on two occasions: simulation modeling and simulation result explanation. When a user requests help for specific problems, the user interface will start a consulting session:

   i) The user interface 'locks' the simulated DFD model.

   ii) The user interface initializes a DDE data link to the simulation expert system (the simulation expert system as the server and the user interface as the client, Interface ⇨ ES), if the link has not been previously setup. The simulation expert system will be executed if it is not currently active.

   iii) The user interface puts the type of the question into the DDE buffer and *pokes* a message: '*I have a question*'.

   iv) Upon receiving this message from the user interface, the expert system initiates a DDE link to the user interface (the user interface as the server and the simulation expert system as the client, ES ⇨ Interface), if the link has not previously been established.

   v) The expert system *requests* a question type and initializes the corresponding knowledge base.

vi) The expert system generates questions for further information from the knowledge base and *pokes*: '*I have a question.*' to the user interface.

vii) The user interface picks up the message from the expert system and *requests* the question. The answer to the question will be gathered from either the data repository or from the user.

viii) Once the answer is available, the user interface pokes a message: '*The answer is ready.*'

ix) The expert system *requests* the answer and continues the inference process.

x) Repeat the steps (vi) to (ix) until the expert system reaches a conclusion or fails. The expert system pokes: '*The answer is ready.*'

xi) The user interface *requests* the answer and 'unlocks' the DFD model.

4. The conversation between the *expert system* and the *simulation subsystem:* This DDE data link is not used in the current HAT design. Nevertheless, the HAT architecture provides the ability to setup dynamic data links between the expert system and the simulation subsystem. Such links are especially valuable for reverse simulation [Wild 91a], in which the expert system monitors the execution of simulation and dynamically adjusts the simulation parameters. A scenario of dynamic data link between the expert system and simulation subsystem in a reverse simulation environment is described as following:

i) The user interface 'locks' the simulated DFD model and feeds the simulation model script including the expected goals to the simulation subsystem through the Interface ⇔ Simulation DDE data link.

ii) The simulation subsystem starts a DDE data link to the expert system (the expert system as the server and the simulation subsystem as the client, Simulation ⇨ ES), if the link has not been previously setup. The expert system will be executed if it is not currently active.

iii) The simulation subsystem puts the simulation goals and constraints into the DDE buffer and *pokes* a message: *'Goal is ready'*.

iv) Upon receiving this message from the user interface, the expert system initiates a DDE link to the simulation subsystem (the simulation subsystem as the server and the expert system as the client, ES ⇨ Simulation), if the link has not previously been established.

v) The expert system *requests* the goal and alternative strategies and setups an *automatic* data link to the simulation subsystem.

vi) The expert system loads the appropriate knowledge base and pokes a message: 'Start simulation.'

vii) The simulation subsystem responds to the message by setting its key statistics (observation windows) to the DDE data buffer and starting the simulation.

viii) Every time new statistics are generated in the simulation, the automatic DDE link will forward the data to the expert system where the data is checked with its goals.

ix) If the expert system finds that the statistics violate the target goals, it will generate a set of new parameters and *poke* a message: '*Stop. Try new parameters.*'

x) The simulation subsystem responds to this message by *requesting* the simulation parameters. Then, the simulation subsystem alters the parameters with new values, adjusts the observation windows and continues the simulation.

xi) Repeat (viii) to (x) until the goals are reached or failed.

The DDE data interface allows the HAT subsystems to be loosely-coupled while keeping sufficient communication among them. The user interface and the simulation subsystem are operated in a 'batch processing' fashion, in which the user interface delivers a job and waits for its completion. The user interface and the simulation expert system have a series of 'question and answer' conversations to solve the pending questions asked by a user.

4.5 The Dynamic Evaluation Subsystem: DFD Simulation

The simulation subsystem is the core to adding dynamic evaluation to the software development process. The key issue of the simulation subsystem is to automatically generate simulation models and execute the models on a simulation engine. Kimbler and Watford [Kimbler 88] give a functional summary of a simulation program generator (SPG) as following:

System Definition: To identify a specific boundary and restriction that a SPG system can handle.

Problem Formulation: To determine the precise problem definition that a simulation model is to address.

Model Development: To develop a model and translate the initially developed form into a form suitable for computer execution.

Data Collection: To take advantage of the data collection facilities provided in simulation languages and allow users to choose what to collect.

Coding: To generate executable code in a simulation language. Some SPG systems may go one step further to address the operating system and environment issues for the simulation runs.

Verification and Validation: To generate error-free code for all possible inputs, to limit users to a specific domain that can be verified by the system, and to continuously validate simulation programs throughout a SPG.

Experimental Design and Production Runs Analysis: To be able to select alternatives and determine a decision based on simulation results.

Documentation and Reporting: To document all the inputs, test results, as well as outputs of a SPG.

HAT limits its scope to simulating a DFD model to reveal the model's dynamic features. Some of the issues of problem definition and model development are dealt with in the user interface subsystem by allowing users to interactively define DFD models. The simulation subsystem focuses on executing a DFD-based simulation model and feeding the simulation results back to the user interface. However, HAT does not address all the issues described by Kimbler and Watford. The above functional descriptions, nevertheless, provide guidelines for the design and implementation of the simulation subsystem and HAT as a whole.

4.5.1 The structure of the simulation subsystem

As described above, the major function of the simulation subsystem is to process the simulation model script from the user interface, generate a simulation model based on

the script, and run the simulation. Figure 4.26 shows the components of the simulation subsystem. A validated DFD model with add-on information necessary for simulation modeling (number of runs, distributions, and so forth) is translated into a simulation model script, which is transferred via a DDE data interface to the simulation subsystem. The model generator parses the script and builds an executable simulation model for the YANSL simulation engine. The simulation results are fed into a result parser where the simulation statistics match each node in the script. The parsed results are then forwarded to the user interface through the DDE data interface.

Since the simulation subsystem is designed as a separate Windows application, independent input and output windows are included as the interface for direct model input and result output. The input and output windows are quite helpful in debugging the simulation subsystem. In addition, with its own input and output capacity, the simulation subsystem can work as an independent simulation system. A user can describe a simulation model in the input window and run the model immediately without the extra steps of loading different modules to compile and run the simulation.

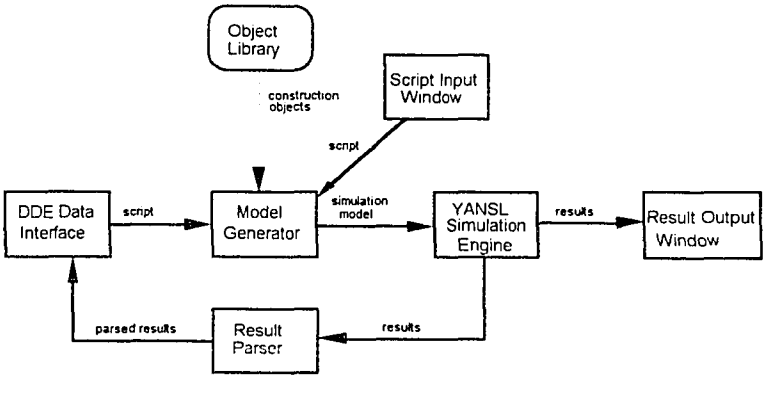


Figure 4.26. The structure of the simulation subsystem

4.5.2 The script language for simulation models

A simulation script language is specially designed to convey information of a DFD model to the simulation subsystem. Within a simulation model script, a DFD model is viewed as a queuing system defined in terms of a YANSL simulation model with source nodes, queues, activities, and sink nodes. The script language is used as an intermediate format that converts DFD objects to the objects in a YANSL simulation model.

The script language has eight sections that have to be completed before a simulation model can be generated. Table 4.2 is an example of a simulation script of a TV repair shop. The syntax of each section can be described as follows:

1) ***Run:{*** *number1; number2; number3;* ***}***

   Defines the number of replications in a simulation experiment (*number1*), the length of warmup period (*number2*), and the length of each simulation run (*number3*). Although increasing the number of replications and the length of the simulation reduces the variance of the system statistics, too many number of runs and too long a single run may result in a waste of computer resources and cause memory overflow. On the other hand, the warm-up period should be long enough to guarantee that the simulation statistics are stable and reliable.

2) ***Distribution:*** *{ type, name, parameter; }*

   Defines the distributions used in a simulation model. All distributions must be declared in this section before they appear in other part of the model.

3) ***Resource:*** *{ type, name; }*

Defines resources used by the queues and activities in a simulation model. Resources must be declared in this section before they appear in the definitions of queues and activities.

4) ***Source:*** *{ type1, type2, name, distribution, start, end; }*

Defines the source nodes in a simulation model. Every line in this section describes the type of transaction (*type1*), the type of branching method (*type2*), name of the node, event distribution, and starting as well as ending times of the event generation.

5) ***Queue:*** *{ type, name, resource; }*

Defines queues in a simulation model. The type, name, and resources used by a queue as defined in each line of script in this section.

6) ***Activity:*** *{ queue, resource, type, name, distribution; }*

Defines the activities in a simulation model. An activity may be directly associated with a queue and a resource. The name and distribution of activity time are described in each line of script in this section.

7) ***Sink:*** *{ name; }*

Defines event exits in a simulation model.

8) ***Branch:*** *{ source, destination, probability; }*

Defines the branches that connect other nodes in a simulation model

Table 4.2. An example of the simulation script for a TV shop

```
// Simulation script of a TV shop

Run:{10; 150; 480;}

Distribution: //Format: type, name, parameters
{
 Exponential, interArrival, 5.0;
 Exponential, inspectTime, 3.5;
 Exponential, repairTime, 8.0;
}

Resource: //Format: type, name
{
 PRIORITY, inspector;
 PRIORITY, repairman;
}

Source: //Format: type1, type2, name, distribution, start, end
{
 TRANSACTION, DET, tvsource, interArrival, 0.0, 480;
}

Queue: //Format: type, name, resource
{
 FIFO, inspectQueue, inspector;
 FIFO, repairQueue, repairman;
}

Activity: //Formation: queue, resource, type, name, distribution
{
  inspectQueue, inspector, PROB, inspection, inspectTime;
  repairQueue, repairman, DET, repair, repairTime;
}
Sink: //Formation: name
{
 finish;
}

Branch: //Format: source, destination, probability
{
 tvsource, inspectQueue;
 inspection, finish, 0.85;
 inspection, repairQueue, 0.15;
 repair, finish;
}
```

4.5.3 DFD model conversion rules

Since a conventional DFD does not contain enough information to convert into a simulation model, additional information is needed to fill the gap [Wild 93]. To simplify the conversion process, several assumptions are made. Although these assumptions may not always hold in reality, they will only have an effect in the model conversion process, but not in the performance of the architecture adopted by the system integration. The author believes that the following assumptions are reasonable to show that dynamic evaluation of DFD models can be done in the context of the HAT architecture.

Assumptions:

(1) A DFD is structurally correct before the simulation.

(2) There is no delay in a data flow.

(3) A data store is a mutually exclusive device that can be shared by different processes at different times. The availability of this device depends only on the sequence of the requests from processes. No preemption of data store resources is allowed and the operation time of a data store is ignored.

(4) All nodes are connected with either probabilistic branches or a deterministic branch, as opposed to many different branching methods in most simulation languages (Cyclic, High/Low, and Conditional).

(5) All the queues in a model only have the FIFO (First-in-first-out) ranking method, as opposed to other ranking methods in most simulation languages (High/Low, Conditional, Random, etc.).

(6) Each external entity represents, at most, a single data in-flow.

(7) Identical discrete events are generated from source nodes during a simulation. There should not be special attributes attached to an event. HAT assumes that all simulation events have a single transaction type: *TRANSACTION* as default.

(8) HAT supports the *replication/deletion* method to analyze steady-state parameters of a non-terminating simulation [Law 91], because the replication method is relatively simple, even though it is more costly in terms of computer resources and efficiency. The overhead caused by the warmup period of multiple replications is not a major concern for the current system.

Conversion Rules:

(1) An external entity or an interface node is a source node if it is a source of a data flow. Further information needed for simulation includes: start/end time for event generation, event arrival distribution, connection assignment information (probability for each branches, if there is more than one connection).

(2) An external entity or an interface node is a sink node, if it is a destination of a data flow. No further information is needed for a sink node.

(3) A DFD process node is a queue-activity pair. Further information needed for simulation includes: activity server distribution and connection assignment information.

(4) A data store has two roles in a simulation model. It is a resource that can be shared by different processes at different time periods. It is also a data flow multiplexer that connects the in-flows with the out-flows. Figure 4.27 shows an example of the multiplexing effect of a data store after a conversion, where each

in-flow connects to each out-flow with multiplied probabilities on each branch. Information needed for model conversion: connection assignment information.

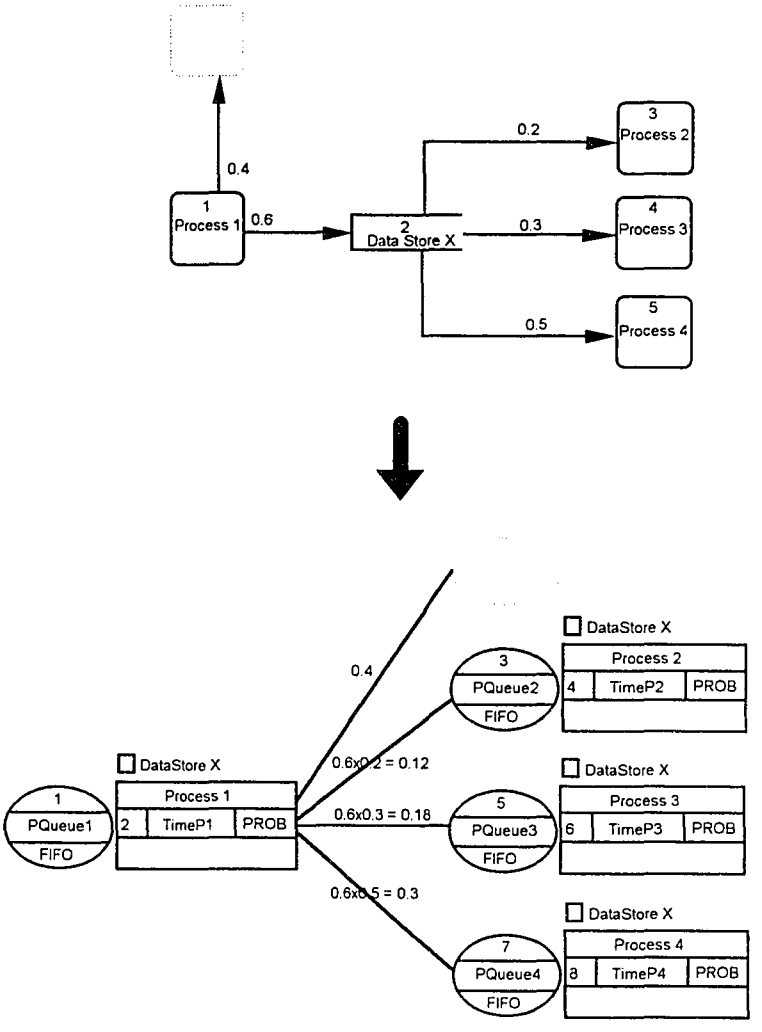


Figure 4.27. The multiplexing effect of a data store after conversion

(5) A data flow corresponds to a branch between the two nodes in a simulation model.

(6) Additional resources can be specified as the pre-requisite for processes. All resources are allocated on an availability basis.

(7) A pseudo sink node should be added to a process to prevent deadlock when:

(ii) a process has only one outflow and the flow is to a datastore.

(i) a process has a loop with a data store whose only outflow points back to the process (see Figure 4.28).

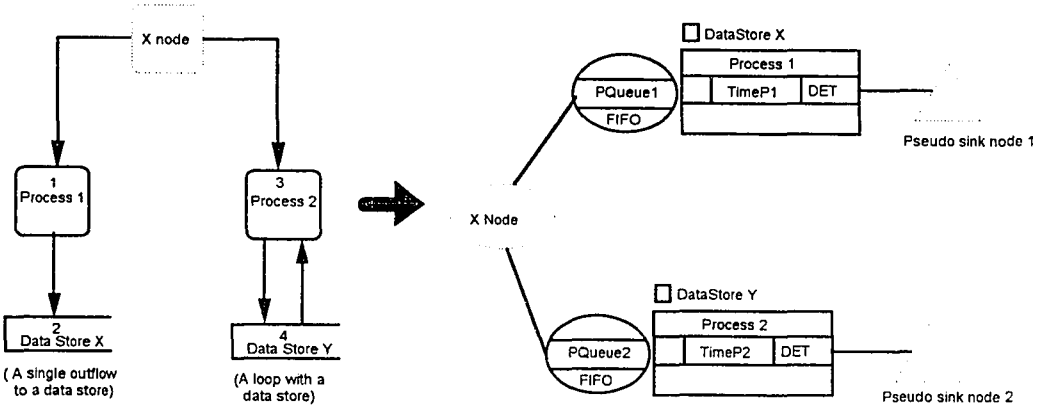


Figure 4.28. Examples of pseudo sink nodes to avoid simulation deadlock

(8) Information necessary to conduct simulation experiments, such as the number of runs, and run length, has to be defined before the conversion.

Considering the above conversion rules, an extended DFD model is converted into a YANSL simulation model as shown in Figure 4.29. There are two additional resources in this DFD model, which are represented with small squares on top of process nodes. The simulation model is an equivalent of the simulation model script shown in Table 4.2.

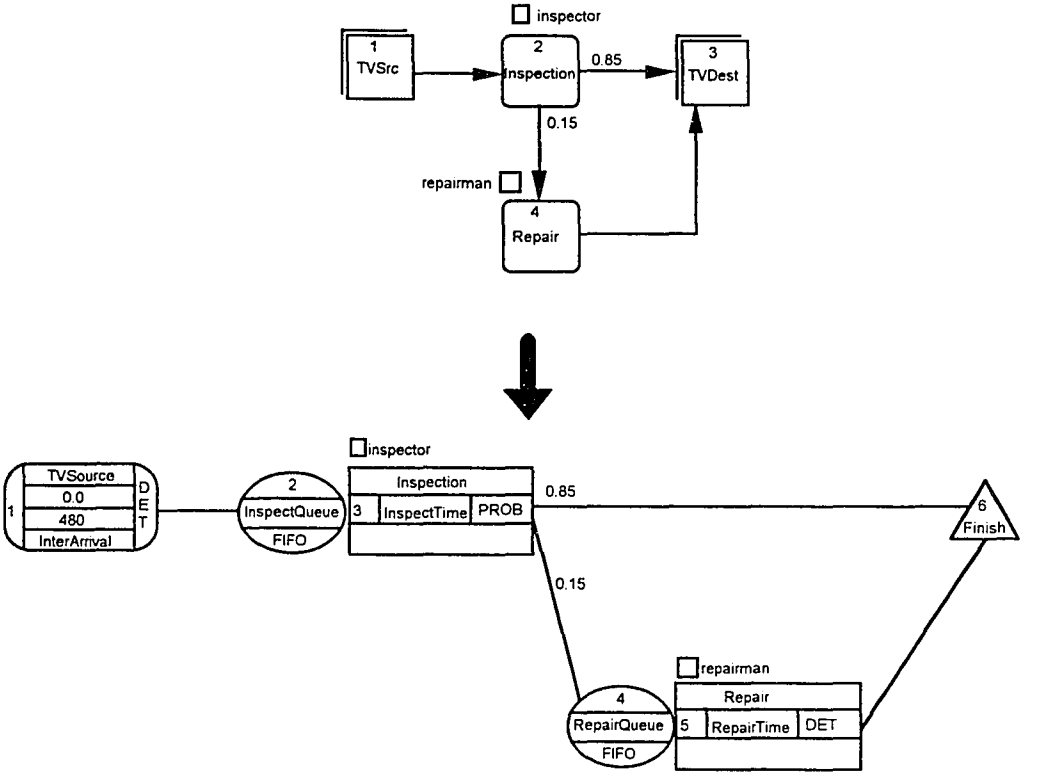


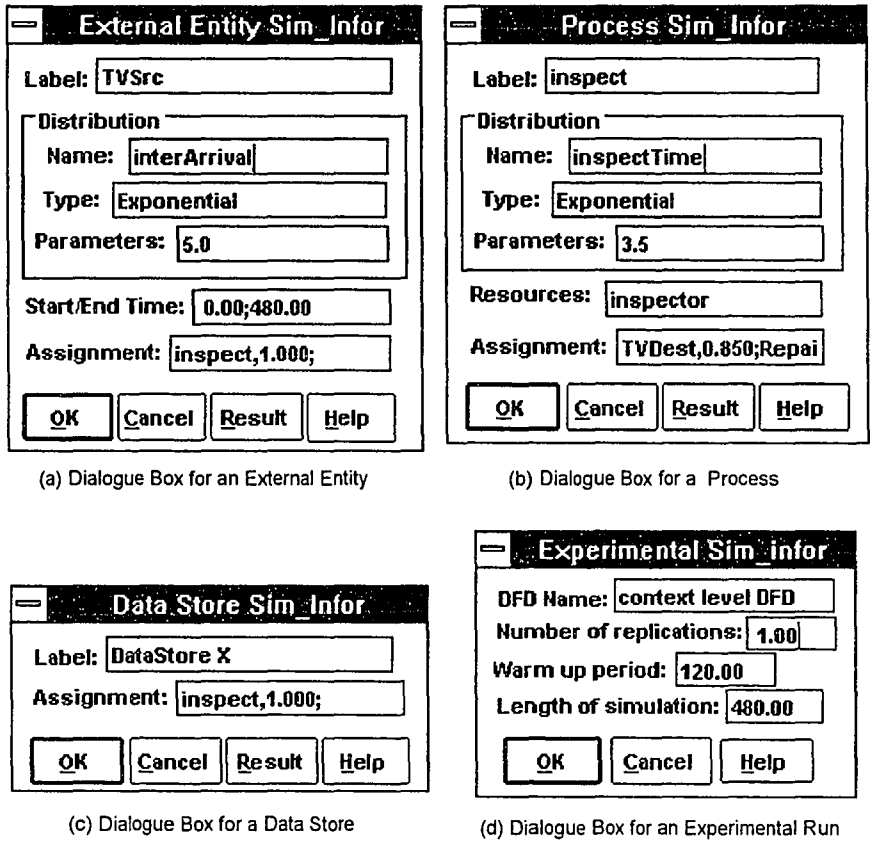
Figure 4.29. An example of converting a DFD model to a simulation model

The DFD model conversion is not a trivial task. It requires the user interface to provide the ability to interactively define the additional information needed for simulation and convert a DFD model into an equivalent simulation script. Therefore, a DFD model converter is added to the user interface.

4.5.4 The DFD model converter

The DFD model converter scans the current DFD model and pops up dialogue boxes for each DFD object in the model to get parameters needed for simulation modeling. Figure 4.30 shows examples of these dialogue boxes. The additional information extends the scope of traditional DFD modeling and encourages users to think more deeply about the system's problems and to observe more aspects of the

system that they are modeling. The 'Result' and 'Help' buttons in the dialogue boxes are used to display simulation results and trigger the intelligent help system.



(a) Dialogue Box for an External Entity

(b) Dialogue Box for a Process

(c) Dialogue Box for a Data Store

(d) Dialogue Box for an Experimental Run

Figure 4.30. Dialogue boxes for simulation parameters

After simulation parameters have been collected, the model converter starts to create a model script based on the current DFD model. Table 4.3 shows the steps used in the conversion process. The converter is an assembly structure with each subclass performing one step in the conversion process. The converter itself is attached to the user interface subsystem.

Table 4.3. Steps to convert a DFD into a simulation model

Steps for DFD _ simulation script conversion

(1) Create the *run* section based on the input from the experimental dialogue box.

(2) Scan all the DFD objects in the current DFD, collects distribution information. and create the *distribution* section of the script.

(3) List all the data stores in the current DFD and the additional resources added to processes and create the *resource* section of the script.

(4) Collect all the external entity and interface nodes with at least one out- going data flow and create the *source* section of the script.

(5) Collect all the external entity and interface nodes with at least one in- going data flow and create the *sink* section of the script.

(6) Create a *queue-activity* pair for each process in the current DFD, with data store and additional resources attached to it.

(7) Create the branch section of the script following the connections of data flows in the current DFD. Data flows to and from a data store need special treatment to create multiplexed branches.

4.5.5 The simulation model generator and result parser

The model generator is a special language interpreter that translates the script language defined in Section 4.5.2 into an executable simulation model. It parses the script and creates the appropriate objects from the YANSL object base. During the model generation process, a parsing table is created that can be used for the result parser. Figure 4.31 is the state transition diagram of the simulation model generator. Each state corresponds to a method that converts a piece of the script into executable objects.

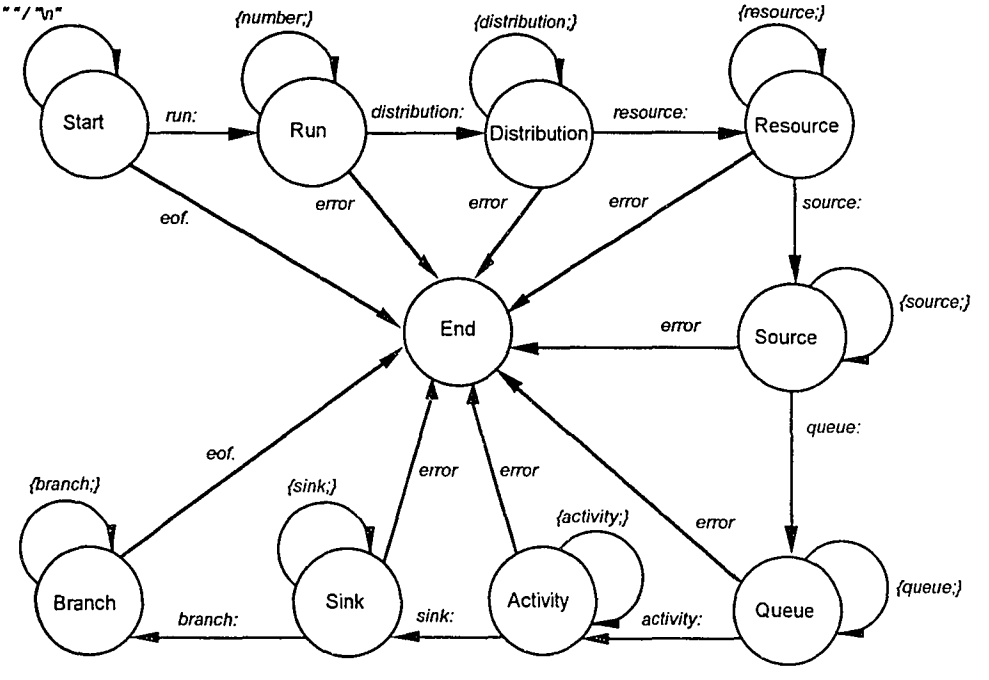


Figure 4.31. State transition diagram of the model script interpreter

After an executable simulation model is generated, the YANSL simulation engine is called to run the model. There is no user intervention or any interruption throughout the simulation. When the simulation is done, the simulation result is fed into the result parser. The output is also sent to the output window of simulation subsystem.

The purpose of the result parser is to match the items of simulation results with the names used in the model script. Since the model generator uses internal expressions to represent an executable simulation model, the simulation results only contain reference numbers to the objects created by the model generator. By referring to the parsing table created during the model generation, the result parser makes the results more meaningful by replacing the reference numbers with their original names. The parsed results will be sent to the user interface and stored in the data repository.

4.6 The Intelligent Help Subsystem

The intelligent help subsystem has two parts: (1) A set of rules to check DFD structure and data balance, and (2) an expert system that provides expert advice to users on simulation modeling and result interpretation.

4.6.1 Static DFD checking rules

The static checking rules focus on the static structures of DFDs and their data balances. A set of connection rules and data balance criteria have been defined and widely used for checking DFD models [Whitten 89]. Table 4.4 is a summary of the connection rules.

Table 4.4. DFD connection rules

	process	data-store	external-entity
process	*	*	*
data-store	*	x	x
external-entity	*	x	x
*: valid connection x: invalid connection			

There are three cases of imbalance in a DFD: the 'black hole', the 'gray hole' and the 'miracle'. A *black hole* is a process or a data store that gets data flow inputs but never has data flow outputs. A *gray hole* is a process or a data store with data flow

inputs insufficient for its outputs. A *miracle* occurs when a process or a data store generates outputs without any input.

Based on these rules, an *BalanceChecker* object is designed to check the DFD structures. Since the connection and balance checking rules are well defined and can be easily implemented in a rule-based expert system, obviously, BalanceChecker can be an object in the expert system. However, considering the dependence of DFD structural information during a checking process, it is better to include a BalanceChecker as part of the user interface subsystem, where access to the data repository is faster and easier. This trade-off makes the implementation of BalanceChecker more compact and efficient.

Data flow imbalances may also occur between two DFDs at different levels. BalanceChecker contains a set of methods that review all the data flow imbalance cases along the DFD tree. All errors are reported to users from the user interface.

4.6.2 The structure of the simulation expert system

In the scope of HAT, the expert system limits its function to simple intelligent advice. For this level of expertise, it may not be as efficient if the intelligent help subsystem is directly embedded in the user interface subsystem. However, simulation-based research involves more expertise than choosing simulation parameters and result explanation. The idea of incorporating an expert system into HAT is to demonstrate that the HAT architecture supports the integration of simulation, expert system and CASE. More sophisticated expert systems can be constructed on this basis. The incorporation of expert systems into simulations have been well described in the simulation research literature [Hill87, Rao88, Taylor88, Mellichamp89, Frankel89, Park90]. The results of these studies are useful for further work based on the HAT architecture. New research may include more expertise in simulation model diagnosis, simulation validation and

verification, simulation experimental design, simulation result analysis, and simulation monitoring.

The HAT simulation expert system has the following operational scenarios to carry out a consulting session after a DFD model has been defined:

(1) A *user sends a request*: A user triggers the expert system by choosing the 'Help' button. The user interface sets up DDE data links to the expert system and a consulting session begins.

(2) *ES and user define problem*: The ES asks questions about the nature of the problem and understands the user's requirements.

(3) *ES 'thinks' about the problem*: and chooses the right knowledge base and matches the user's requirements with conditions and rules.

(4) *ES provides advice*: After several iterations of step (2) and (3), the ES will advise the user on how to define simulation models and how to run the simulation.

(5) *The ES and the end user engage in a dialogue about the results:* After the simulation is finished, the ES can be accessed again to check the significance of the results and interpret the results with terms understandable to the user.

Similar to the simulation subsystem, the simulation expert system gets requests from the user interface and starts a consulting session via its DDE data interface. The expert system will determine what type of questions a user asked and load either the modeling rule base or the explanation rule base. As a separate Windows application, the simulation expert system has its own user interface, through which a user can create and modify rule bases. This 'local' expert user interface is very useful in developing and debugging the rule base.

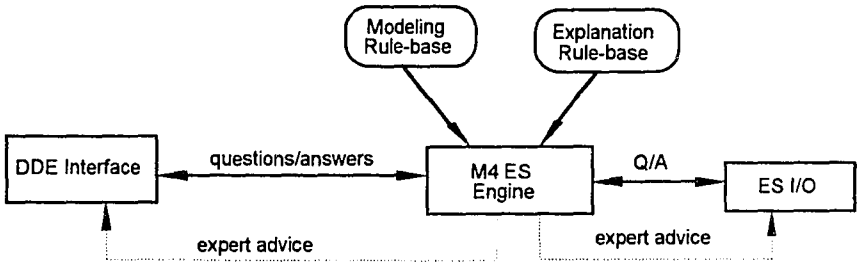


Figure 4.32. The structure of the simulation expert system

As indicated in Figure 4.32, the expert system engine has two modes: the *stand-alone* mode and the *server* mode. In stand-alone mode, there is no DDE connection to any other application. The expert system gets commands and displays outputs using its own user interface. In the server mode, the expert system works as a server for other client applications through DDE data links. All the questions and answers are directed to the client applications. Since M4 is a general-purpose rule-based expert system shell, there is no specific restriction that ties the simulation expert system to the HAT environment. The simulation expert system can be used for other consulting tasks simply by loading different rule bases.

4.6.3 The modeling rule base

Creating a correct simulation model is a complex task that requires clear understanding of domain problems and rigid validation and verification processes. Mastering the expertise of the whole simulation modeling process is beyond the scope of the HAT expert system. The intelligent help system only tries to answer the following questions:

(1) What distribution should be chosen?

(2) What should be the parameters for the distribution?

(3) How many replications should be run and what should be the length of the warmup period?

Currently, there are only three distributions included in the YANSL prototype environment: Exponential, Uniform, and Normal. More distributions can be derived from the YANSL pseudo random number generator. Figure 4.33 is the decision tree that determines the selection of a distribution and its parameters. This tree can be expanded as more distributions become available.

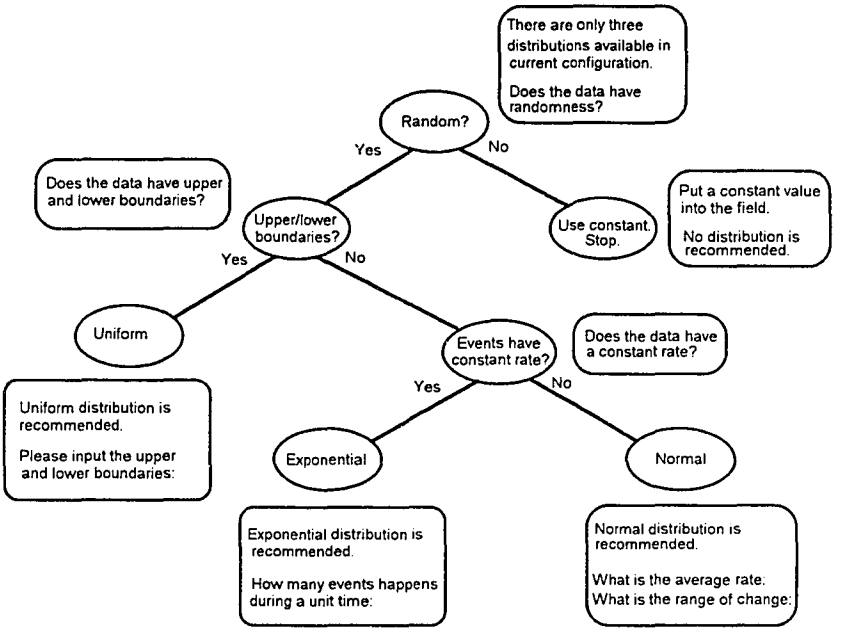


Figure 4.33. The decision tree for distribution selection

Determining the number of replications, the warmup period, and the run length requires more expertise than distribution selection. Statistical estimation of these parameters needs pilot simulation runs and intensive statistical analysis, which is worthy of a separate study of expert system applications in simulation. The current HAT expert system offers only minor assistance in the design of a simulation experiment based on pilot simulation runs. Instead of complete assistance, a set of empirical rules is used. These rules take the following principles into consideration:

(1) The difference between simulation length and warmup period should be adequate to allow a sufficiently high number of observations per simulation run to secure the significance of simulation statistics.

(2) The warmup period should be long enough to cover the transition period.

(3) An increase in the number of replications or the simulation length should narrow the confidence intervals of output parameters. However, too much overhead of the simulation is inefficient and may cause system overflow.

With these principles in mind, the empirical rule requires that the simulation length be at least 4 times longer than the warmup period. The default number of replications is 10 and the default simulation length is 8 hours or 480 minutes. These empirical rules are not sufficient to always generate statistically significant simulation experiments. However, they are sufficient to test the HAT architecture. More sophisticated expert system are possible in future research.

4.6.4 The result explanation rule base

A typical simulation output analysis includes testing the significance of a simulation experiment. As part of this analysis, the following types of questions are asked –

Is the warmup period long enough to cover the transient period of the simulation?

Does the simulation have sufficient replications and length to fulfill the IID (Independent Identical Distribution) assumption?

Are the variances of the estimated random variables kept within expectation?

If these conditions are not met, new experimental plans should be recommended accompanied by a new set of simulation runs. It is clear that the simulation output analysis and selection of simulation experimental parameters (step (3) of last section) are iterative and closely related. The final evaluation is based on a satisfied output analysis.

An expert system that masters the expertise of a significant output analysis needs several years to develop and requires a rule base of several hundred rules [Taylor 88]. Developing such a complex expert system is clearly too much for this project. Nevertheless, previous studies have shown that an expert system with sophisticated simulation expertise is doable in a personal computer environment.

HAT is focused only on the structural aspects of expert system integration in a simulation environment. It omits the critical output analysis part and assumes that the output from the simulation is final. This will produce biased system estimations and cause the simulation results to be insignificant. Additional work needs to be done to include the expertise of output analysis.

With limited output explanation functions, the explanation rule base provides only a verbal explanation of the simulation results (mean, variance, utilization, time in system, time in queue, etc.). It also includes several empirical rules to determine if a process is under-used or becomes the bottleneck of a system.

CHAPTER 5 IMPLEMENTATION ISSUES

As in the case of many software development projects, there are many ways to implement a concept or a software architecture. This is especially true in the development of PC applications for which the tools and environments are evolving very rapidly. This chapter is dedicated to the discussion of issues concerning the implementation of the architecture described in Chapter 4 and the lessons learned through this project.

5.1 Windows Programming Environments

HAT aims to help novice personal computer users. Because of the popularity of Windows-based PC systems, Microsoft Windows is a natural choice for the HAT target environment, As shown in Figure 5.1, several Windows programming environments (PLUS , ToolBook 1.5, SmallTalk for Windows 1.0 and Microsoft C 7.0 etc.) are examined based on two major criteria: the visualization (graphical user interface, user friendly design environment) and the processing power (efficiency, effectiveness, and functionality). Borland C++ 3.1 and Visual Basic 3.0 are chosen because for the following reasons:

(1) *C++ is powerful in representing object-oriented concepts and structures.* Though SmallTalk is more object-oriented than C++, C++ is more efficient and capable of representing basic object-oriented concepts. During a search for the HAT implementation environment, the author found that an application developed in the current version of SmallTalk for Windows takes large amount of disk space and runs slower than an equivalent C++ application. On the other hand, although other Windows programming environments, such as PLUS, ToolBook, and Visual Basic, have strong support for the user interface design,

they are not sufficient to support the complicated class structures and object-oriented concepts required by HAT.

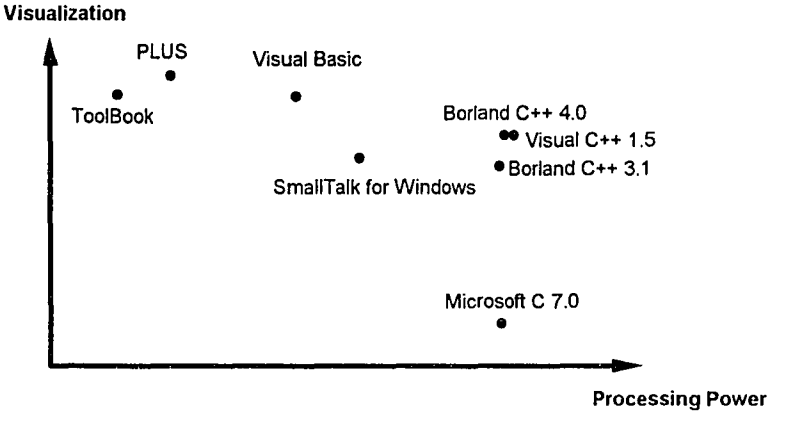


Figure 5.1. Comparison of different Windows programming environment

(2) *C++ is closely linked with the Windows operating system.*

The Windows operating system is based on the C language. Various ready-to-use C++ Windows application frameworks and application packages are available. These C++ packages directly interact with the Windows system kernel through the Microsoft SDK (System Development Kit) and API (Application Programming Interface), and provide many add-on features for Windows programming. In addition, C and C++ are also very popular languages on mainframe operating systems, such as UNIX. With the acceptance of object-oriented concepts, C++ has become the *de facto* standard object-oriented programming language in the software industry.

(3) *C++ is resilient to change.*

As an object-oriented programming language, C++ encapsulates data and methods within an object. There will be little ripple effect incurred by

modification. This feature fits the exploratory nature of HAT development, where changes are constantly made as the system evolves.

(4) *Borland C++ provides a user-friendly development environment.*

The Integrated Development Environment (IDE) provided by Borland 3.1 was the only real Windows-based C++ development environment at the time HAT was designed. The Borland IDE integrates editing, compiling, linking and debugging utilities. It also includes tools to visually create graphical resources.

(5) *C++ has third-party supports.*

Because of inheritance and reusability in C++, many software companies have dedicated their efforts to developing general purpose software packages that can be used for graphical design, data processing, knowledge processing, data structure development and data base operation. There are a large number of reusable libraries written in C++. The availability of these libraries has saved time, reduced the overall risk of failure, and improved the system quality.

(6) *Visual Basic is easy-to-use and compatible with existing software package.*

Visual Basic (VB) is chosen because it has the ability to embed libraries developed with other languages. VB is powerful for user interface design and rapid prototyping. It also supports cross-application communications, such as DDE and OLE. VBX (Visual Basic eXtension) is a special run-time Dynamic Linking Library (DLL) format that can be added to the Visual Basic Workbench as a special control object. A VB application can embed mission-critical VBXs developed in more efficient languages, while remaining the user friendliness of a Visual Basic. M4, the expert system used in HAT, is developed in C. The expert

system package provides a ready-to-use VBX, that makes it easy to integrate with a Visual Basic application.

Windows programming environments are developing so fast that new compilers with improved Windows programming supports, such as Borland 4.0, Visual C++ 1.0, and 1.5, became available before this project was finished. These new products do not negate the initial choice of Borland 3.1 and Visual Basic 3.0. On the contrary, the new developments are indications that the choice of Windows environment and C++ language provide a good foundation for further research and enhancements.

5.2 System Integration

Three subsystems in HAT are developed as separate Windows applications, namely the *user interface subsystem*, the *simulation subsystem*, the *simulation expert system*. The user interface subsystem is the most complicated subsystem in this project; the hypertext editor, the graphical editors, and the data repository are developed. The other two subsystems are based on reusing existing software packages.

An incremental approach is used for system integration. Naturally, the development of a DDE communication framework is the first step toward system integration, because all the subsystems are interconnected through DDE data links. As indicated in Figure 5.2 (1), three functionally identical applications were created, each having bi-directional DDE communication capacity. Each application has a general application handler that contains the information of the names of *server*, *client*, *topics*, and *items* of its DDE interface. An application handler is also an internal data channel to a specific application. An application with its own handler can be directly 'plugged' into an application handler 'slot', just like plugging an I/O card into a PC mother board.

The triangle structure of the DDE interfaces was tested in different modes: *poke*, *request*, and *automatic*. Testing on this triangular back-bone helps to identify and consolidate the problems with DDE communication between two C++ applications, as well as a C++ application and a Visual Basic application. Since none of the HAT subsystems were involved at this stage, the integration focused on DDE communications among three simple and functionally identical applications so that the complications and side effects of non-DDE factors are reduced to a minimum.

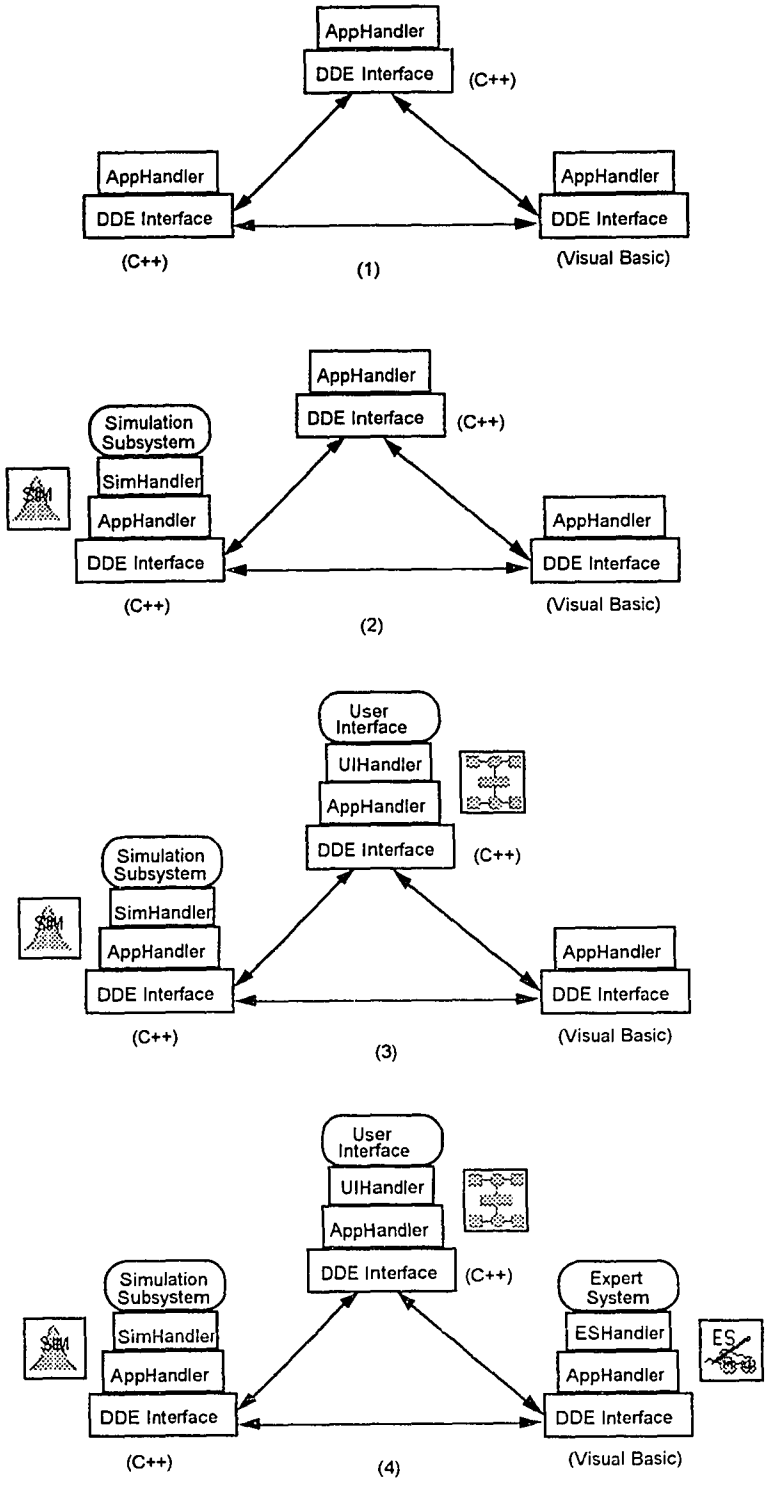


Figure 5.2. Steps in systems integration

The next step in integration is to add a simulation subsystem (see Figure 5.2 (2)). The simulation subsystem has its own handler - SimHandler, which is a descendent of AppHandler and has the same behaviors with AppHandler. SimHandler handles special data transfers required by the simulation subsystem and passes the data onto AppHandler. The relationships of each subsystem and the DDE-AppHandlers are described in Figure 5.3.

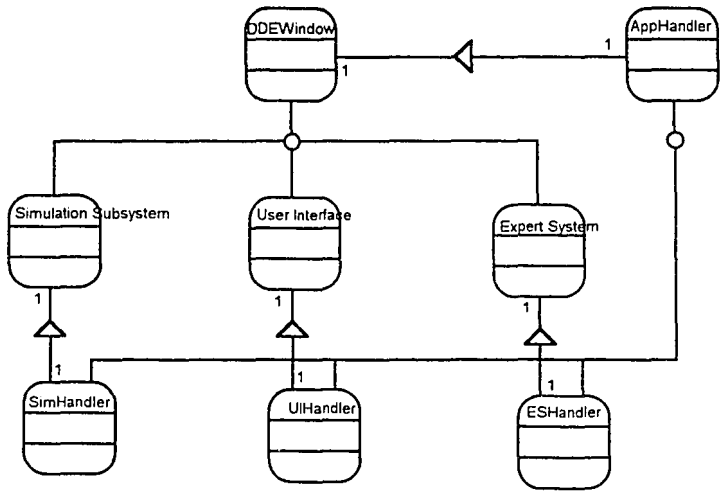


Figure 5.3. Class structures of subsystems and their handlers

Similarly, special handlers (UIHandler and ESHandler) are created for the user interface and the expert system, through which the user interface and the expert system are plugged into the triangle DDE communication backbone (see Figure 5.2 (3) and (4)).

When a new application is introduced to the DDE backbone, the handlers and communications are tested to insure that the integration is working as expected. When the three subsystems are connected, an integrated testing procedure is performed to guarantee that all the subsystems are working independently as well as cooperatively.

5.3 System Testing and Evaluation

System testing involves both validation ('Are we building the right product?') and verification ('Are we building the product right?'). Systematic program testing is a complicated process that involves careful planning and test case generation. It is important to understand that testing can never show that a system is correct. It is always possible that undetected errors exist even after the most comprehensive testing. Program testing demonstrates the presence of errors but not their absence [Sommerville 89].

Two methods are often used for program testing: '*black box*' testing and '*white box*' testing. 'Black box' testing does not require a tester access to source code or understand the program being tested. In contrast, 'white box' testing relies on a tester's knowledge about the code and the structure of the program being tested. In general, 'black box' testing is suitable for end users who know what they want, but have no knowledge of the details of coding, and 'white box' testing is often used by programmers who know all the program details.

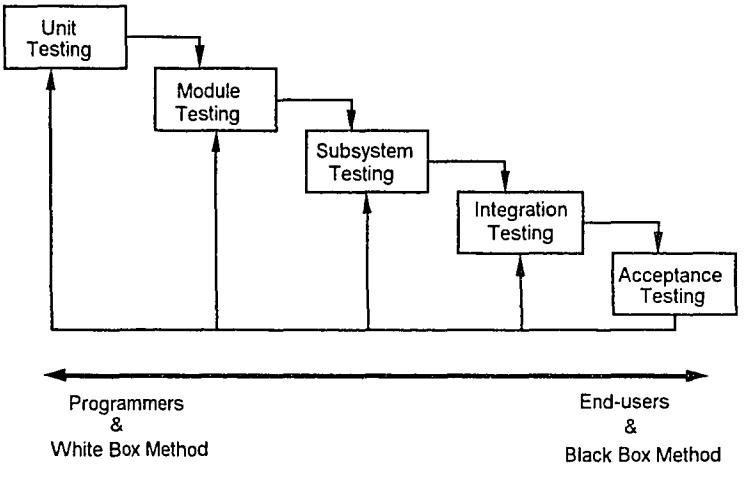


Figure 5.4. The stages and the factors involved in testing

As indicated in Figure 5.4, project testing may have different stages from the simplest components to a complex whole. The 'Black box' method is often used to find

out where a fault is, and the 'white box' method is used to determine how a fault happens and the method to fix it. System developers are an inseparable part of the system testing process. They know how to carry out 'white box' testing to fix bugs and improve system performance. However, programmers are not the best system testers. Psychologically, they consciously or subconsciously try to avoid the presence of errors and the destruction of their creations, which is contradictory to the purpose of system testing. Outsiders, especially potential end users, should be employed in the testing process to overcome the programmer's psychological bias.

The validation of this project focuses on the functionality of HAT to see if the tools included in HAT are sufficient for simple systems analysis tasks. For two consecutive semesters, HAT has been used as courseware for systems analysis and design classes for undergraduate MIS students. Through these field tests, it has been shown that HAT is sufficient for simple systems analysis cases in these classes.

System verification ranges from unit testing to integration testing and includes both 'black box' testing and 'white box' testing. With 'black box' testing, a tester performs different operations and matches the system output or behavior with the user's expectation. Once an inconsistency or a bug is found, 'white box' testing method is used with the help of debugging tools to examine the details of each module for a line by line, value by value analysis of the reasons for the problem. Since a Windows application receives many different events, messages, and their combinations, many testers should be invited to test alpha and beta versions of a system to remove errors. The students who used HAT for their homework assignments helped to detect many bugs and design faults that were ignored by the HAT developers. Even though HAT has not been a bug-free product, the extensive verification has made HAT correct and robust for most of the required operations.

An extensive evaluation of a user interface design is a very expensive process. It requires the support of cognitive scientists, such as psychologists and graphical design specialists. The evaluation involves designing and carrying out a statistically significant number of user experiments and is only economically possible for very large systems development projects.

Obviously, this project does not have the resources to conduct an extensive statistical evaluation of the HAT user interface. The size of this project does not justify such an extensive study either. Nevertheless, some simple empirical user interface evaluations were carried out among a group of undergraduate MIS students. Two methods were used during the evaluation: observation and a questionnaire.

The survey result of 16 MIS students shows that all students have PCs at home and most of them use HAT both at home and school. This result shows that HAT has reached its goal as a personal front-end CASE tool to help novice users leaning the basic systems analysis techniques. As shown in Table 5.1, the students generally give positive comments on the HAT user interface and its performance. Most of them think HAT is easy to use and can learn it within 2 hours, which is a much shorter learning curve than other educational CASE tools, such as EXCELERATOR. The survey also shows that some students have problems with the scheme of hyperlink creation and navigation facilities. These are the clues that HAT should be further improved.

Table 5.1. The survey result of 16 HAT users

HAT helps in understanding DFD and ERD.	1 (Disagree)	2	3	4	5 (Agree)	Total Answers
	0 / 0%	0 / 0%	3 / 18.8%	9 / 56.3%	4 / 25%	16
HAT is easy to use.	1 (Disagree)	2	3	4	5 (Agree)	
	0 / 0%	0 / 0%	4 / 25%	12 / 75%	0 / 0%	16
HAT improves understanding of relationships between process and data modeling.	1 (Disagree)	2	3	4	5 (Agree)	
	3 / 20%	3 / 20%	6 / 40%	2 / 13.3%	1 / 6.7%	15
How long did it take to learn HAT?	< 2 hours	2-5 hours	1 day	> 1 day		
	10 / 62.5%	3 / 18.8%	1 / 6.3%	2 / 12.5%		16
Did hyperlinks between narrative text and graphical objects help learning?	Very helpful	Helpful	Confusing	No effect		
	1 / 6.3%	13 / 81.3%	1 / 6.3%	1 / 6.3%		16
How do you like the multi-window interface?	Easy to use	Acceptable	Get lost			
	13 / 81.3%	3 / 18.8%	0 / 0%			16
How do you like the graphical tools?	Like it	Acceptable	Don't like it			
	10 / 62.5%	6 / 37.5%	0 / 0%			16
How do you like HAT navigation tools?	Like it	Acceptable	Confusing			
	8 / 50%	8 / 50 %	0 / 0%			16
Was it easy to create hyperlinks?	Easy	Acceptable	Hard to learn			
	3 / 20%	12 / 80%	0 / 0%			15
Does the help facility help?	Very helpful	Acceptable	Not helpful			
	5 / 31.3%	8 / 50%	3 / 18.8%			16
Is the reporting / printing function sufficient?	Sufficient	Acceptable	Not sufficient			
	5 / 31.3%	9 / 56.3%	1 / 6.3%			16
Is the response time satisfactory?	Yes	No				
	11 / 68.8%	5 / 31.3%				16
Where do you use HAT?	Lab.	Home	Both lab&home			
	1 / 6.3%	2 / 12.5%	13 / 81.3%			16
Do you have computer at home?	Yes	No				
	16 / 100%	0 / 0%				16
Do you use HAT at home?	Yes	No				
	15 / 93.8%	1/ 6.3%				16
Have you used Windows application before?	Yes	No				
	16 / 100%	0 / 0%				16
What type of PC do you have?	PC 486	PC 386	PC 286	PC 8088		
	9 / 60%	5 / 33.3%	0	1 / 6.7%		15

5.4 Alternative Integration Strategies

The DDE is not the only method to integrate the HAT subsystems. There are different alternatives that may yield the same result, such as direct coupling, pipelining, file sharing, dynamic link library (DLL), OLE, and networking.

Direct coupling of simulation and expert systems with CASE is difficult, and error-prone, because the subsystems themselves have been very complicated applications. Direct coupling may also require special internal structures that fail to reuse existing packages. Nevertheless, a well-designed direct coupling integration can be more efficient than other alternatives.

Pipelining is often used in single user, single tasking systems like DOS, or a multi-tasking system like UNIX. In a single tasking system, applications can only run one at a time. Pipelining only transfers data once an application is finished, which does not meet the requirements of HAT. Although MS Windows is a multi-tasking system, it does not provide the powerful tools for multi-tasking and concurrent programming as UNIX. Application integration through multi-tasking pipelines and process communication is difficult under MS Windows.

File sharing allows multiple applications have access to the same file through 'single write - multiple read' or 'multiple write - multiple read' file access control conventions. File sharing is a good way to share data among different applications. However, it does not provide sufficient controls mechanisms for the applications to coordinate synchronized communications. HAT requires well-coordinated conversations among its subsystems. Therefore, file sharing technique is not suitable to support HAT.

A Dynamic Link Library (DLL) is a library format supported by Windows. A conventional library is linked to an executable program at linkage time. As part of the

executable program, a conventional library may have multiple occurrences in different program executions. A DLL can be shared by multiple applications and linked to the applications at run-time. There is at most one copy of a DLL in memory in contrast to possible multiple occurrences in a conventional library. As a result, the DLL is a way for applications to share and reuse common functions in Windows. All the libraries used in HAT (Tools.h++, ObjectGraphics, YANSL, and M4) can be transformed into DLLs, which will significantly reduce the size of the HAT executable module and save memory. However, using DLLs for system integration is still a kind of direct coupling strategy. A subsystem represented in DLLs cannot work as independent applications. Moreover, programming and debugging in DLLs are often more difficult than that of conventional programming.

As a superset of DDE, OLE can perform every DDE function. In addition, OLE supports a more sophisticated data format and automates object linking and embedding. It is possible to integrate the HAT subsystems with OLE instead of DDE. However, DDE is more efficient and can be tailored for more user-dependent communication among applications. DDE has the advantage of managing multiple items in a single conversion, which often requires multiple conversation links in OLE. DDE has been chosen over OLE as the linkage method for HAT subsystems because multiple data items are exchanged depending on user choice. More recent versions of OLE provide OLE Automation and OLE Controls and will eventually become universal communication links that integrate Windows applications [Pleas 94].

There are several successful commercial products for system integration using 'application suites' and 'cross application platform' concepts of 'pluggable' applications that include Borland Office 2.0, Lotus SmartSuite 2.0 and Microsoft Office 4.0. These systems integrate a word processor, spreadsheet, database, electronic mail, as well as

graphical and presentation tools into a single platform that allows different applications to share data and functions through DDE, OLE, and DLLs while keeping the independence of each application. In light of these successful systems, HAT is an application of the 'cross application platform' concepts to CASE, simulation and expert system. The HAT architecture can be seen as a platform for visual interactive CASE/Simulation systems.

Integrating applications on different machines over a computer network is outside the scope of this project. Nevertheless, this research indicates a direction for further study. There have been network DDE and network OLE applications that apply the same DDE and OLE principles over a local area network. An extension of the HAT architecture based on the network version of DDE and OLE would be applicable for distributed CASE/Simulation integration.

DDE and OLE concepts can also be combined with DEXTER model structure to integrate diverse software packages. DEXTER model is a reference model that captures the important abstractions in a wide range of hypertext systems [Halasz 94]. Figure 5.5 shows a scenario of DEXTER node integration based on a DDE / OLE network. This scenario can be seen as a generalization of HAT architecture. Each application in this scenario is constructed as a DEXTER node. The run-time layer is the user interface that present information (text, graphics, image, voice, and video) in a uniformed format for the integration. The storage layer manages the information storage and retrieval. In addition, the storage layer also has DDE / OLE communication capacity that sends information back and forth to other DEXTER nodes. The DDE / OLE media can be constructed based on either networking or stand-alone DDE / OLE protocols. The within-component layer is application specific, where details of each application are implemented. The three layers of a DEXTER node are relative independent and can be

designed separately. The run-time layer and within-component layer can be plugged into the storage layer through presentation interfaces and anchors. The storage layers of different DEXTER nodes become the backbone of the integration. Different DEXTER nodes can be designed in different environments as long as they can communication with DDE / OLE protocols.

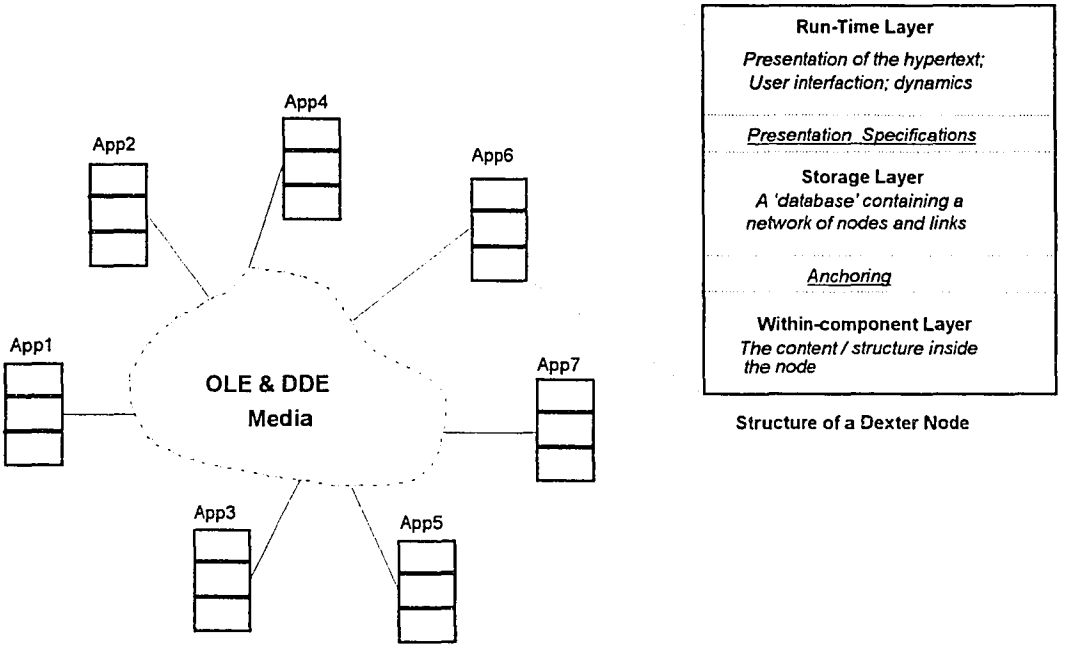


Figure 5.5. A software integration scenario based on DEXTER model

5.5 Lessons Learned Through The Implementation of HAT

Although C++ is a powerful object-oriented language with strong links to the Windows operating system, it is not without problems. Programming in C++ requires that programmers be concerned with the details of object management. The power of C++ comes at the cost of additional responsibility and risk for programmers. Unlike interpreted languages, such as Visual Basic and PLUS, C++ needs extra time for compilation and linkage. It also lacks the system support of object management and error protection found in Visual Basic and PLUS.

There are two problems that require special attention in C++ programming: the 'gone and forgotten' problem and the 'memory leak' problem. One of the features of object-oriented programming is dynamic binding, in which objects are created and associated with methods at run-time. Because of the lack of object management in C++, it is the programmer's responsibility to monitor memory use. If an object is created and never deleted, it will become memory garbage that occupies system memory even after the application is terminated. When sufficiently bad, memory leaks may result in total operating system failure. The programmer is responsible for 'garbage collection' by deleting objects from memory when the objects are no longer needed. However, premature deletion or repeated deletion of an object will also cause an operating system failure. This phenomenon is known as the 'gone and forgotten' problem. Therefore, it is critical for programmers to manage objects in a timely and organized fashion. Much of the object management in C++ relies on the programmer's experience, debugging tools and well-designed object structures.

Other lessons learned throughout this project are:

(1) *Reuse existing C++ classes and C++ code.*

The inheritance feature of C++ made software reuse become a reality. Software reuse will shorten development time and improve system reliability. However, special care must be taken to insure the quality of class library packages purchased from outside sources. Because of the additional difficulties in reading and editing the code of others, bug detection is difficult.

Another potential problem with software reuse is updating and technical supporting from package providers. This is especially true in new and rapidly evolving systems, such as Windows. Many development environments are not

fully backward compatible. Application packages based on an old environment often need major changes to take advantage of a new environment.

Two third-party C++ libraries are used in HAT: *Tools.h++* and *ObjectGraphics*. Tools.h++ uses ANSI standard C++ to provide a cross-platform SmallTalk-like collection classes for data structure design. Tools.h++ has little difficulty with the change in different Windows programming environments. On the other hand, ObjectGraphics is closely associated with the Windows GDI (Graphic Device Interface) and the Borland OWL 1.0 (Object Windows Library) Windows Application Framework to provide graphical programming support. A program developed with ObjectGraphics is not portable to other GUI (Graphical User Interface) environment. It is also very sensitive to changes in the Borland C++ programming environment.

(2) *Take advantage of Windows Application Framework (WAF) and User Interface Toolkit (UIT).*

The basic tools for Windows programming are Microsoft SDK (System Development Kit) and API (Application Programming Interface). However, programming directly with the SDK and API is very tedious and error prone. There are several Windows Application Frameworks including the Object Windows Library (OWL) from Borland and the Microsoft Fundamental Classes (MFC) of Microsoft. These are the most popular C++-based WAFs built on top of the SDK and API.

UITs are also included in the Windows programming environment. Microsoft AppStudio and the Borland WorkShop provide graphical tools to visually design a user interface.

WAFs and UITs provide a friendly user interface and result in more efficient Windows programming. While gaining ease in Windows programming, programmers lose the flexibility of direct access to the SDK and the API, because WAFs only provide services for standard window operations. Nevertheless, programmers can always use direct SDK and API calls from a WAF to regain flexibility, when necessary, because the SDK and the API are special C functions that can be called from any C++ applications. For instance, several direct SDK calls are made in the HypertextWindow of HAT to implement the hypertext display, while the HypertextWindow itself is a descendent from the TFileWindow class of OWL.

(3) *Organize a program around events and messages.*

Fundamentally, HAT contains a hypertext-based user interface. Object-oriented GUI design is an event-driven process. Objects and their associated methods are organized in response to events from the WIMP user interface and messages from one window to another. Event-driven programming requires the identification of the events and messages handled by an object and the construction of appropriate methods. This is in contrast to traditional function-driven programming, where execution sequences are pre-defined inside a program . In addition, there are no apparent execution sequences in an event-driven program.

Event-driven programming is effective in handling graphical user interface events and messages, where users may choose various event combinations regardless of the sequence. However, end user freedom poses a 'threat' to programmers. Programmers have to consider all the possible events, messages and their combination effects in order to define the event-handler accordingly.

Because of the additional requirement for event handling, a graphical user interface design is much more difficult than that of a command line user interface. Programmers often rely heavily on GUI design tools, debugging programs, and GUI design experience to cope with the increasing demand for graphical interface design.

(4) *Rely on good interactive debugging tools.*

Because of the complexity inherent in graphical user interface design, it is imperative that powerful debugging tools be used throughout the development process. A good on-line debugger will help programmers visually examine program execution flows, values of variables, stack and heap usage, class structures, and source code. The original Borland C++ 3.1 only includes a DOS-based debugging tool, which is not sufficient for complex Windows programming. The recent versions of Borland C++ 4.0 and Visual C++ have greatly improved interactive debugging tools.

During the development of HAT, investments in debugging and testing tools exceeded the expense of programming environments. Several third-party debugging tools were employed to assist in finding program bugs, logical design errors, as well as memory leaks, pre-mature deletion, and repeated deletion problems. Without these debugging tools, it would be difficult, if not impossible, to develop the system.

(5) *Rely on well designed class structures and documentation.*

C++ offers programmers great flexibility for implementation, but does not enforce rules for good system design. The quality of the source code as well as the whole system lie in the hands of programmers. Using C++ does not

guarantee the benefits of object-oriented techniques, instead, methods for object-oriented analysis and design must be followed. If global and public variables and methods are not restricted, and class structures are not well defined, a C++ program can be as bad as any other poorly designed program.

HAT uses the Coad-Yourdon method for system analysis and design. The project benefited from carefully designed class structures. Due to the exploratory nature of this project, systems details were not clearly defined at its inception. Thus, object-oriented methods helped to define the system components and permitted the system to evolve through many modifications.

(6) *Follow examples and intelligent help facilities.*

Examples help significantly in Windows programming, especially for beginners. Given the level of sophistication in Windows programming, a beginner is often overwhelmed by its complexity. A good example can provide an entry to Windows programming. There are many functions and calling conventions in the Windows environment and it is difficult to use them correctly and efficiently all the time. Reference to examples or source code from previous projects can save a lot of time.

Some Window programming systems include intelligent help facilities to reduce the burden on programmers. Microsoft AppWizard, ClassWizard, and Borland AppExpert, ClassExpert are good examples. These facilities let programmers visually create graphical interface and generate skeleton code, message map, and event handlers for the application. Also, the intelligent help facilities save a lot of effort by specifying correct program structures and connecting the program with graphical resources.

(7) *User interface design is difficult.*

The friendliness and 'easy-to-use' features of graphical user interfaces come at the expense of higher requirements for coding and debugging. Myers notices that 40-50% of the code and run-time memory are devoted to interface functions in some application [Myers 89]. The ratio is getting larger as more complicated graphical user interfaces emerge.

The graphical user interface is the focus of HAT. More than 60% of the source code is dedicated to the manipulation of windows and graphical objects. Even though HAT is not a bug-free product, it has taken several months of debugging to make it run smoothly. The author believes that the HAT experience is common to all GUI designs that one cannot expect a bug-free GUI after a few runs. Given the powerful tools for programming and debugging, GUI development is still not an easy programming job. A substantial amount of time has to be allocated to design, testing and debugging.

(8) *System testing is iterative and requires involvement of many testers.*

A software system often needs several rounds of modifications to satisfy its requirement. Some software bugs are formidable and require time and many rounds of testing to be isolated and fixed. There is a chance that modification and bug-fixing may introduce new bugs and inconsistencies that causes new problems. As a result, system testing is an iterative process and is never over.

In addition, GUI user interface testing requires not only significant amounts of time, but also the involvement of many testers, especially outside testers, to explore the different ways the user interface is touched, clicked, moved, and

navigated. The author has observed that programmers are often trying to prove the correctness of their system and follow certain testing patterns. They are not effective testers. On the contrary, an outside tester often does not have anything to prove or any pattern to follow. Observing how outside testers interact with the system often lead to uncovering more bugs and ill-defined functions that were never noticed by the system developers.

CHAPTER 6 CONCLUSIONS

This project presents an effort to improve the performance of upper-CASE by introducing a new interface design and dynamic evaluation. The concept of the integration of hypertext and simulation with the traditional structured analysis tool is novel in CASE research. This research is inspired by the current trends of both software engineering and simulation research: (1) the need for more user involvement, (2) the need for IS dynamics, (3) the increasing level of system integration, (4) the potential benefits of a visual and an interactive environment, (5) the development of an intelligent and user friendly interface, and (6) the need to improve system portability and flexibility.

6.1 Contributions

This project chose hypertext to improve the user interface and simulation to enhance system model evaluation. The integration of these techniques in a multiple window environment provides a set of flexible and compact systems analysis tools. The primary contributions of HAT are:

(1) *Improvement of user involvement:* One of the primary objectives of this research is to improve user involvement. There are apparent necessities for more user involvement [Ives 84]. The advances in hardware technology, especially micro-computer technology, broadens the base of users who willingly participate in IS development. HAT takes advantage of the state-of-the-art microcomputer environment and techniques to provide an interactive, visual, hypertext-based user interface. It provides an easy method for users and system developers to understand the non-linear structure of system models and encourages users to employ structured techniques. Subsequently, HAT makes the DFD and ERD techniques more understandable and closer to the end users. The end users and

the system developers may communicate better because they are using the same tools and following the same systems analysis philosophy.

A preliminary user survey shows that HAT does help novice user understand the structured analysis methods and improve communication among instructors and students.

(2) *Complement approaches to JAD problems:* The basic JAD problem is to find ways to improve communications with end users and understand their requirements better. JAD principles involve the introduction of structures and formats for 'how to run a design meeting'. Although it is powerful and effective to organize a design meeting and convey information among users and system developers, current JAD lacks the ability to evaluate overall system dynamics. In addition, JAD introduces a set of new expression methods, from different structured analysis methods. This may result in some conversion problems later in the analysis process. HAT makes use of structured techniques (DFDs and ERDs) to communication with users and estimate system dynamics directly. This approach pushes DFD and ERD tools toward end users. It will convey the information user requirement more effectively.

(3) *Introduce IS dynamic analysis to CASE:* Businesses and organizations are dynamic by nature; the more complicated they become, the more dynamic feedback they need. So far, CASE tools have not included dynamic analysis as a routine task. It has been proved that dynamic analysis can improve system performance estimation and decision making, especially in a noisy and turbulent environment [Warren 92, Wild 91a]. HAT introduces an automatic DFD simulation procedure of DFD models that provides a macro estimation of system

performance at the early stage of the SDLC. In previous studies, simulation model generations are either manual or limited to automatic transfer of pre-defined models. HAT provides greater integration and stronger dynamic linkages.

(4) *Better system integration:* HAT uses DDE data links for dynamic data transfer among the user interface, the simulation package and the expert system – an integration that previous studies have not implemented. In the testing systems of Eddins [Eddins 90] and Wild-Griggs [Wild 91a], no automatic model generation was implemented. Warren's study [Warren 92, 93] developed a prototype that incorporates simulation with CASE. It supported automatic simulation model generation but did not have an interactive user interface to input DFD models - the models have to be pre-defined in another CASE tool before the automatic model generation and simulation. HAT integrates a full-featured hypertext-based DFD and ERD user interface and automates the process of DFD simulation.

(5) compact and flexible structure: HAT is developed in the Windows environment with C++ and limits its function to system analysis only, which results in:

(a) The efficiency of C++ and the limited functionality make HAT more compact and less costly than other CASE tools. HAT can be used on a lap-top PC in the field for system analysis. Users may also have their own copies of HAT and work on their own problems at home or at the office - this represents one more step to bring CASE tools out of the computer laboratory and closer to end users.

(b) The DDE link structure makes HAT very flexible. The HAT subsystems are 'pluggable' and can be assembled at run-time, because they are independent applications under Windows supporting DDE

communication. HAT can be easily down-sized to use the interface alone for systems analysis, while the simulation subsystem and the expert system serve other purposes.

(6) *Provide a framework for other studies:* The concepts proposed in this research may find applications for purposes other than IS development:

(a) DFD is a very powerful modeling method. It can be used for the modeling of office routines and work flows. These models do not necessarily lead to an implementation of computerized systems, but rather are descriptions of how businesses are handled in an office. Once these models are described in DFDs, users can use HAT to evaluate the dynamic features of these models and weight different alternatives.

(b) HAT can also be a simulation environment, where simulation models can be described in DFDs and HAT can generate the simulation model automatically. The intelligent help system may provide the same services to aid simulation modeling and simulation result explanation.

(c) The dynamic links among subsystems can be improved to support stronger interactions. Instead of using 'batch processing' mode to handle simulation models, run-time interaction can be added to exchange simulation parameters and update rule-bases. This will result in strong dynamic links among the subsystems that can be used for reverse simulation [Wild 91b] and simulation animation. Since the construction of HAT is object-oriented and based on an open structure, the enhancement toward stronger dynamic links based on current HAT architecture is applicable [He 94b].

6.2 Limitations

HAT is limited to the support of the systems analysis stage of the SDLC with DFD and ERD. Currently, HAT is a stand alone system. There have not been connections of any kind with other CASE tools proposed in this project, though such connections are possible and desired for CASE integration [Chen 92a and b]. User query analysis has been limited to keyword search technique. Natural language processing capacity has not been incorporated with the user interface.

The file-based system of HAT is not sufficient to handle large and complex projects. Moreover, users cannot access the same project at the same time. HAT has no facilities to support cooperative IS development.

There is not enough interaction in the 'batch processing' simulation mode. Users do not have interactive control over the simulation once it is started. The simulation expert system is only a simple prototype that cannot undertake real simulation tasks.

The simulation package used in this project is only a prototype version. A lot of modeling and statistical features common to most simulation languages are not included in YANSL. A more advanced object-oriented simulation package is expected to improve dynamic evaluation. In addition, the dynamic evaluator simulates only one DFD model at a time. The correlation among different DFDs is not included. There is a potential to expand the scope of dynamic evaluation to include all information system models in a project.

The description of a process node in a DFD is nothing more than simple text, which limits the capacity for further analysis. Formal or semi-formal expressions of a process may improve the quality of the analysis, such as decision trees, decision tables, and pseudo code.

The current hypertext editor is based on links and pointers, which is convenient and practical for a prototype system. In the long run, a standard hypertext engine with SGML (Standard Generalized Markup Language) parser is preferred to handle large hypertext documents. SGML is more flexible and powerful in terms of hyperlink creation and navigation, with which different hyperlinks and hyper-views of the same hypertext documents can be created automatically by a hypertext engine.

6.3 Future Research

This project reveals more interesting questions on visual, interactive systems analysis and simulation environments than it solves. The following are potential topics for each subsystem that can be improved in future studies:

The user interface:

- Interface with other CASE tools that cover later stages of the SDLC.
- Improve the interactivity of the user interface by providing simulation tracing, stepping and animating capacities to follow the details of system dynamics.
- Extensive user behavior study on effectiveness of the HAT user interface and dynamic evaluation functions. The subject of the study should be end user or novice system developers. The purpose of this study is to see how much this system will help novice users learn systems analysis techniques and how effective they use it for their own problems.

The data repository:

The incorporation of an Object-Oriented database system, like RAIMA Object Manager, for data repository management so that multiple users can

share information of the same project concurrently. This will also pave the way for collaborative development based the HAT architecture.

The simulation subsystem:

- Use better simulation packages. Because YANSL is a primitive simulation package, it is not powerful enough to provide more sophisticated modeling and statistic capabilities. A verified commercial simulation package will serve better within the HAT architecture.

- The conversion rules used in this project is YANSL specific. Some of the assumptions and rules may not be suitable for other simulation languages. More study is needed to create a generalized set of conversion rules for non-FIFO and complex queuing systems.

- Larger simulation coverage. Expand the scope of simulation to include multiple DFD models of different levels. Give users the freedom to choose the level of details they expected to investigate.

The expert system:

- Take advantage of the achievement of current AI and simulation application studies and enhance the simulation expert system to cover more simulation tasks.

- Expand the expert system in HAT to include intelligent hypertext operations that recognize link patterns and group link clusters. Use special marker languages, such as SGML, HTML, to represent hyperlinks so that the expert system can manage the semantics of the text and create hyperlinks automatically for different user levels.

The interface for dynamic data exchange:

- Upgrade the dynamic data exchange interface to advanced OLE operations and enhance the current data interface functions and provide more data services.

- Develop network DDE and OLE data interfaces and lay the ground work for group system analysis tools.

In the long run, extensive research of the integration of hypertext, simulation and expert systems along the lines of the HAT approach will include:

- Extension the HAT architecture toward an object-oriented systems analysis toolkit that enforces object-oriented concepts and provides hypertext and simulation support.

- Expansion to a reverse simulation environment that includes a visual graphical interface, an expert system, and a simulation kennel. In the same fashion, a visual interactive simulation (VIS) system can also be constructed.

- Expansion to a visual interactive decision support tool by using different graphical modeling methods and providing dynamic evaluations. Such a tool can be incorporated into an management information system for planning and process re-engineering. Decision makers can use a drawing board to change the models and the system will give statistics of possible impacts.

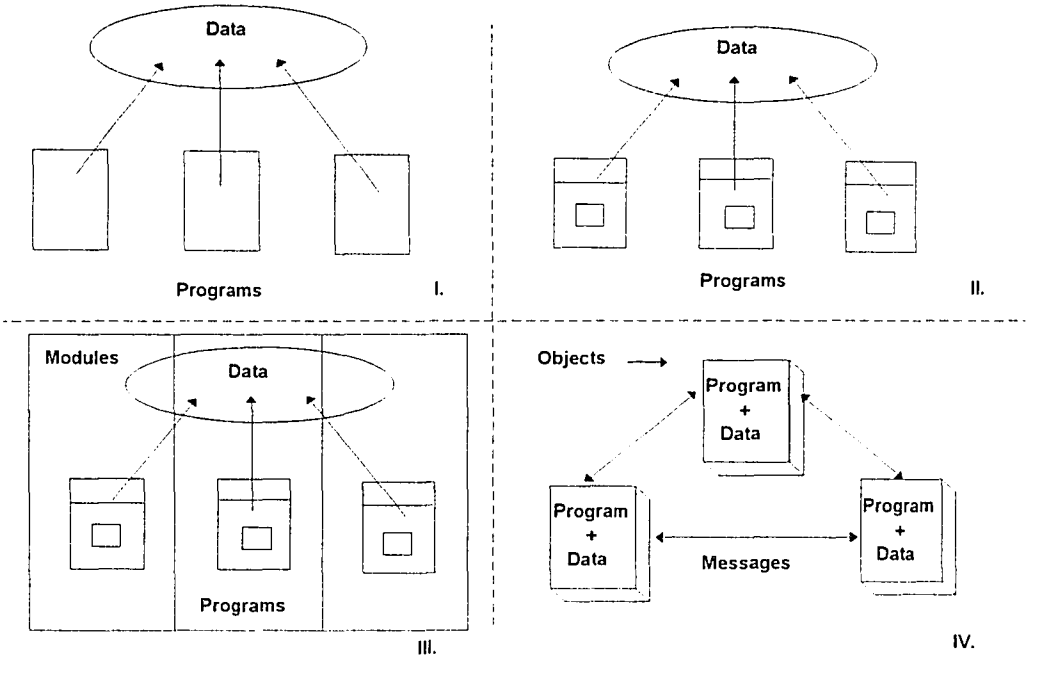
APPENDIX A. AN OVERVIEW OF OBJECT-ORIENTED TECHNIQUES

Object-oriented Analysis (OOA), Object-Oriented Design (OOD), and Object-Oriented Programming (OOP) are the most promising techniques of the 90's [Yourdon 92]. It is expected that OO Database and OO Operating Systems and all kinds of OO applications will eventually replace and enhance traditional products. What is OO? Why is it so powerful?

What are Object-Oriented Approaches?

Software development has gone through several different stages before the concepts of OO became practical. The following figure shows an outline of these different stages.

History of Data Processing Models



Stage (I): All data is exposed to programs. A program has access to any data without any restriction. Programs are closely related by common data. Ripple effects (changes in one program cause changes in other program) are very strong. (i.e. BASIC, Assembler and FORTRAN II)

Stage (II): Data is defined as 'public data' and 'local data'. Each program may maintain its own data that is only accessible to itself. This effort reduces some ripple effects, but programs are still tied to public data. (i.e. FORTRAN 77, ALGOL 60)

Stage (III): Programs are defined as modules with their own local data and algorithms. Data sharing among modules is minimized. However, data and algorithm are still treated separately. (i.e. Pascal, ALGOL 68, C)

Stage (IV): Data and programs are tied together as an *object*. There is no direct data sharing among objects. Objects communicate by sending messages and providing services. This is the so called *Object-Oriented approach*. (i.e. SmallTalk, C++)

Grady Booch [Booch 90] defined OO approaches as:

> *To view the world as a set of autonomous agents that perform some high level behavior ...*

This definition emphasizes that objects are 'autonomous agents'. They have their own 'high level' behavior and are relatively independent from the outside world. An object provides services upon requests (messages) from its information clients. Each object may also send messages to request services from its information providers. In any event, outsiders have no control over how the services are provided and why the information requests are generated.

Each object has its own data members (attributes) and methods (procedures that provide services designated to the object). The object owns its attributes and methods throughout its life cycle and has the following basic features:

(a) *Abstraction* is to ignore those aspects of a subject that are not relevant to the current purpose, in order to concentrate more fully on those that are [Coad 90]. For an object, abstraction means both *data abstraction* and *procedural abstraction*.

Data abstraction means that data is well organized in structures so that it can be treated as a single type or types to feed into operations (procedures). *Procedural abstraction* means a well-defined operation that can be treated as a single entity, even though it may combine many other operations.

In an object, attributes are data abstractions of all the properties that the object has. Although they may be as simple as a Boolean variable or as complicated as combinations of other objects, the object treats it as a single attribute entry to its methods. Similarly, a method provides unique service to its users. The users treat the method as a single service even though it may contain complex operations.

(b) *Encapsulation* is also called 'information hiding' which means that each object should keep its design decisions as locally as possible. An object should not reveal its attributes and the inner-structures of its methods to the outside world unless it is necessary.

Encapsulation is fundamental to keep each object autonomous. Because of encapsulation, objects are loosely coupled. The connections among them are simply message paths. Direct controls and data access among objects should be avoided.

Encapsulation also limits the ripple effect of modifications. Changes in one object may have little effect on the other objects as long as messages and services (interfaces) are unchanged.

(c) *Inheritance* is the ability to propagate characteristics from ancestors. Objects are usually organized in family trees with more general objects on the top and more specific objects at the bottom. A child object inherits properties from the parents. In addition, a child object can add new properties and change those properties inherited from its parents. A parent object may hide some of its properties as 'private' so that nobody else will be able to use or inherit these private properties.

Inheritance makes it easier for software reuse. Useful properties of old designs can be inherited and enhanced by new designs. Therefore, resources can be focused on solving new problems rather than replicating old designs. Inheritance also gives new ways to use third party's software. Some generic and fundamental operations can be inherited from standard software package to make our software production more effective and efficient.

(d) *Polymorphism* enables a function name to be shared within a class hierarchy allowing each class in the hierarchy to implement the action in a manner appropriate to itself. The particular version of a polymorphic function to be executed is determined at run time. Dynamic binding supports the feature of selecting the code to perform a particular function at the time the function is invoked.

(e) *Dynamic binding* means that attributes and methods of an object are generated and binded together at run time. The attributes and methods belong to this specific object throughout its life cycle.

An object is described in terms of a *class* that contains all the descriptions of attributes and methods for the object. When an object is generated from its class at run time, an instance of the class will be created. This process is somewhat similar to assembling a circuit based on its schematic.

Dynamic binding guarantees that all the objects generated from the same class will have the same behavior. Since each object is autonomous, any two objects cannot interfere with each other, even if they are generated from the same class. Therefore, each object will keep its own dynamic status at run time while having the same basic features as the objects generated from the same class.

Pros and Cons of Object-Oriented Approaches

Object-Oriented approaches bring new strategies for software development. They break through the barrier between data and procedures and bring a new vision to every stage of information system development. Some of the major advantages are listed as following:

(1) *Provide a frame work that supports basic methods of human cognition.* Behavioral studies reveal that human recognition has three basic patterns: (a) To identify an object and its attributes; (b) To distinguish a whole object as its components and (c) To distinguish among different objects. OO approaches follow the similar patterns to identify and define objects to make a system more understandable. OO approaches also make analysis and design focus on objects and their relations rather than specific details of data and procedures.

(2) *Focus primarily on problem space understanding.* Because the feature of abstraction, designers can concentrate on the certain level of abstraction at one time and forget other irrelevant facts.

(3) *Combine data and process models into an intrinsic whole.* All the techniques before OO separate data and process models. Although these approaches may let the designers focus on one of the models at a time, different representations of data and process models often cause misunderstanding and ambiguity during system development. The Dynamic Binding feature allows data and procedures being created and destroyed at the same time and eliminate all the side effects of separate data and process models.

(4) *Encourage reuse of software components.* Because the feature of inheritance and encapsulation, it is easy to reuse some components of previous designs or standard package from third parties and reorganize them into a new system.

(5) *Reduce development risk and expense.* Because of software reuse and inheritance, most mature designs can always be reused and inherited in new products. Therefore, the new products will be more reliable and cost less than from scratch.

(6) *Leads to systems that are more resilient to change.* Because of information hiding, changes to the objects will limit the ripple effect to a minimum and make the whole system more flexible.

(7) *Allow to accommodate families of systems.* This is an obvious result of inheritance. New products inherit basic features from previous versions and add new features.

Although there are many advantages in using OO, it is not a panacea to all software development problems. There are some limitations in implementing an OO system:

(1) *OO approaches may not be quite suitable for number-crunching jobs.* OO is good at user interface design, process control and communication, database management, and information system modeling. It may not be as efficient as some existing methods in dealing with number-crunching jobs because OO approaches require more resources to operate.

(2) *Transition and start-up costs are high.* OO approaches are relatively new when compared with various information system technologies. Most installed systems are not OO. The movement to OO based systems is not easy.

(3) *Training requirements.* Most software engineers are not trained in OO concepts and a long learning curve is typical in making the transition to OO.

(4) *Performance overhead.* OO approaches need extra resources to handle messages and pass services among objects. Management of objects may also take extra CPU time and memory.

Object-Oriented Analysis: Coad-Yourdon OOA Method

Object-Oriented techniques are not only new programming techniques, but also a set of methodologies that support the entire systems development life cycle. System analysis is the first step in system development. Object-Oriented Analysis (OOA) can be defined as following:

> *OOA is a method of analysis that examines requirement from the prospective of the classes and objects in the vocabulary of the problem domain.*
> *~ Grady Booch ~*

> *OOA is the process of identifying and defining classes and objects.*
> *~ Peter Coad ~*

There is no standard for OOA. Some proprietary OOA methods have been used for system development. Among them Coad-Yourdon Method [COAD 90] and Booch Method [BOOCH 91] are popular.

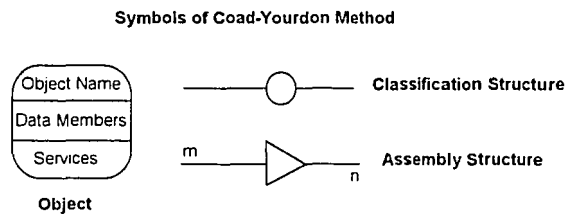
The Coad-Yourdon OOA Method emphasizes the definition of objects and their classes. It has five basic steps: *Identifying Objects, Identifying Structures, Defining Subjects, Defining Attributes, and Defining Services.*

Identifying Objects: Considering requirement specification, system analysts will identify concepts that represent the basic components in the requirements. Objects can be

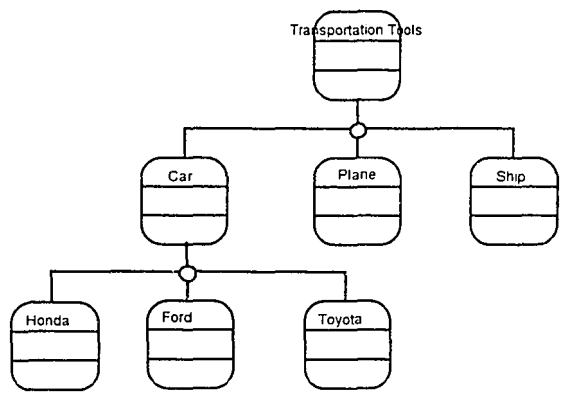
the name of a problem domain, a picture, or an object in the real world. Once an object is identified, it is normally named with a noun or adjective and noun.

It is not possible to identify all the objects necessary for system development at this stage. However, it is vital to identify major objects in the requirements. Detailed and specific objects can be identified as the decomposition proceeds.

Identifying structures: An object in the real world is often a complex whole of smaller objects. Class structures are necessary to represent this complexity. The Coad-Yourdon Method defines two types of class structures: *Classification Structure* and *Assembly Structure.* The process of Identifying Structures is to categorize objects and find the underlying detail.

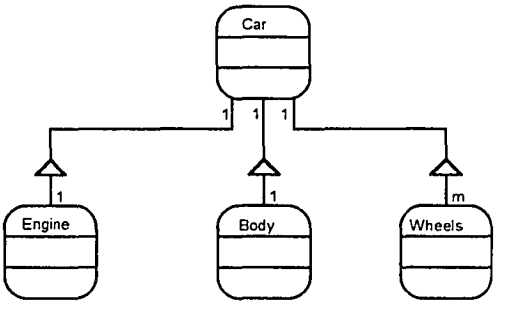


Classification Structure represents a generalization of a type of objects. For example, 'transportation tool' is a Classification Structure of 'Car', 'Plane' and 'Ship'; 'Car' is a Classification Structure of 'Honda', 'Ford' and 'Toyota'. Different levels of Classification Structure construct a concept tree with general concepts on the top and specific concepts at the bottom.



Example of Classification Structure

Assembly Structure represents an aggregation of different components in an object. For example, A 'Car' is composed of 'Engine', 'Body', 'Wheels' ... A set of Assembly Structure will form a component tree with a general object on the top and different levels that reveal different details of the general object.



Example of Assembly Structure

Identifying Subject is to control how much of a model that analysts consider at one time. This is to classify related objects into different layers so that analysts can focus on specific objects at one time. Generally speaking, a subject should be limited to no more than seven or so objects (Miller's Law) [Coad 91].

Identifying a Subject is different from Identifying Structures. A Subject depends on requirements. It is the specific logical relation among objects for a given requirement specification. Structures depend on concepts. It is the intrinsic relation among objects that are independent from the requirements.

Defining Attributes: Attributes are data elements contained in an object that describe its characters and status. Several objects may share some common attributes. General and common attributes should be put into the objects on top of a class structure tree and specific attributes should be put together with specific objects.

Defining Services is to define what methods should be included in an object to provide the required service. There can be three different kinds of services:

Fundamental services such as initialization of an object, monitor and change of status, and basic calculations.
Book-keeping services such as keeping object history and basic sequences.
Event Response Services such as response messages from other objects.

Object-Oriented Design: Coad- Yourdon OOD Method

As the second stage of system development, system design is going to decompose the result from system analysis toward implementation. Object-Oriented Design (OOD) can be defined as following:

> *OOD is a method of object-oriented decomposition and a notation for both logical and physical as well as static and dynamic models.*
> *~ Grady Booch ~*

OOD is a process of adding details for system implementation, including human interaction, task management and data management details.
~ Peter Coad ~

In addition to their OOA methods, P. Coad and E. Yourdon propose a set of OOD methods. It contains four basic components: *Problem Domain, Human Interaction, Task Management,* and *Data Management.*

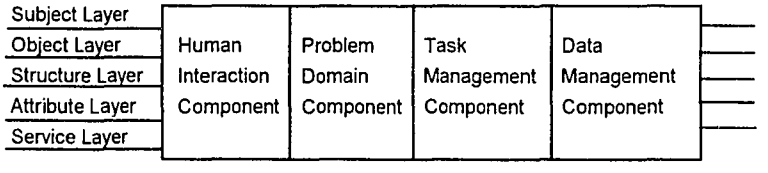
The **Problem Domain component** is directly inherited from OOA and refined in OOD. Coad-Yourdon methods provide a set of symbols that make OOA and OOD use uniform representation. This will avoid some of the inconsistency conversion of different representations. System analysts should strictly follow the results from OOA and not add objects that are outside the problem domain.

The **Human Interaction Component** captures how humans will command the system and how the system presents information to users. Human interfaces are vital for system acceptance and system performance. OO techniques are good at interface design. Humans can be treated as objects that send messages and requests for services. Computer objects are triggered by these messages and provide services accordingly.

The **Task Management component** manages various kinds of tasks and processes in a system. This component is specially important for large applications where multiple tasks are running simultaneously. Task Management will identify the coordinator, the event-driven tasks and clock-driven tasks to synchronize them.

The **Data Management component** provides uniform data service to other objects. Data can be stored on different devices with different formats. In an OO system, everything is treated as an object, including the storage device. The Data Management component is an OO shell that wraps around the data storage to provide unified service to other objects in the system. Other objects receive data from the Data Management regardless of where it is stored.

Most system analysis and design methods in the past have different representations. Conversion from one representation to another may cause information distortion and misunderstanding. Coad-Yourdon OOA and OOD methods use a set of uniform representation that make analysis and design intrinsically connected. The transition from analysis to design can be very smooth. The Coad-Yourdon OOA and OOD methods can be summarized as following diagram.



Coad-Yourdon OOA and OOD method summary

APPENDIX B. EXAMPLES OF SYSTEMS ANALYSIS WITH HAT

B.1. Analysis of a TV inspection workshop operation: balance of workflows

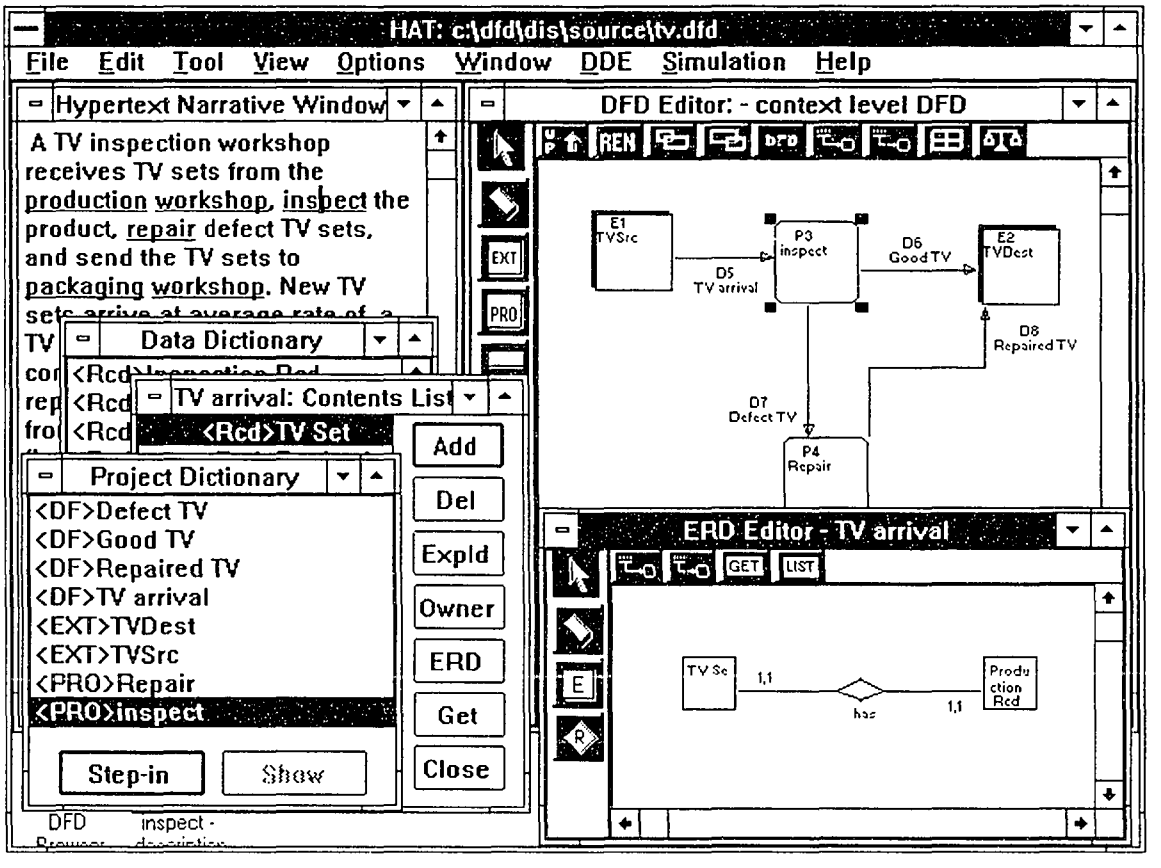
1) The problem description:

A TV inspection workshop receives TV sets from the production workshop, inspects the product, repairs defective TV sets, and sends the TV sets to packaging workshop. New TV sets arrive at average rate of a TV set every 4 minutes. Quality control (QC) group reports that 85% of the TV sets from the production line are flawless, and 15% need to be repaired. The QC report also shows that inspection of a TV set takes about 2 to 3 minutes. A defective TV set is repaired every 15-30 minutes.

Assume that there are one group of inspectors, another group of repairmen in this inspection shop, and all the defective TV sets can be repaired. What is the throughput of this workshop? How busy are the workers? What can be done to improve the system?

2) The system models and hyperlinks:

Based on the information above, a DFD model is created. Along with the DFD model, data models that describe the data flows are also created. The following diagram shows a snapshot of HAT interface that describes the TV inspection workshop.

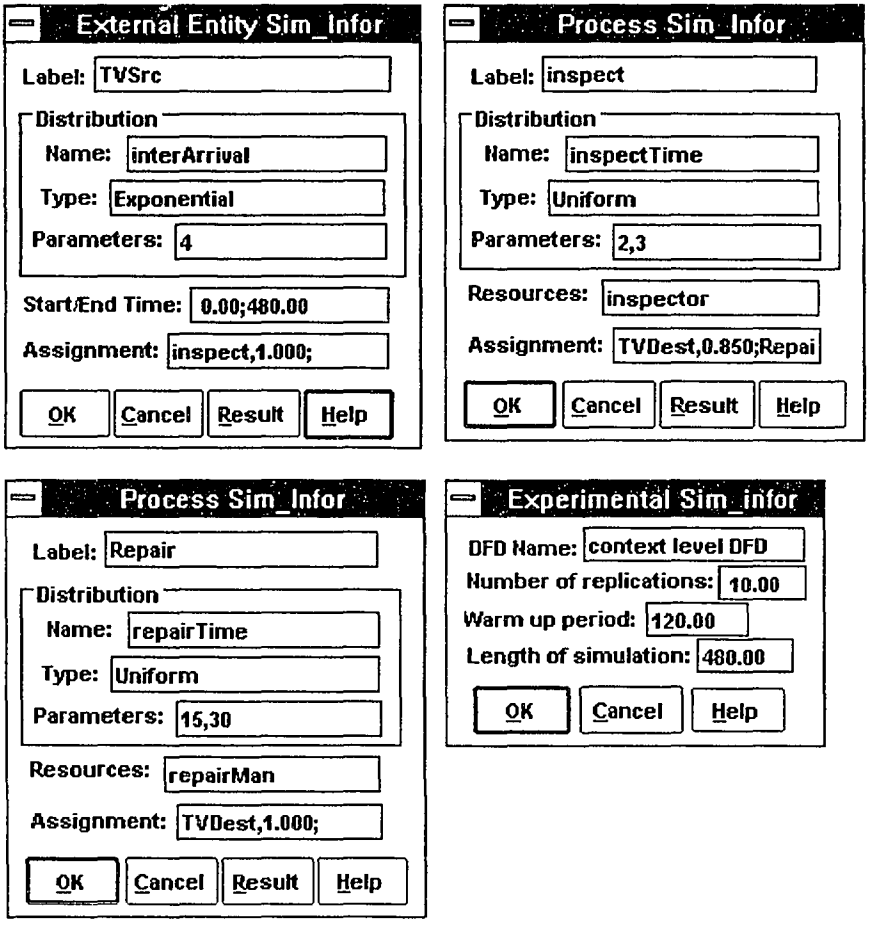


3) Simulation parameters:

By double clicking on a DFD object, a user can access to the description of that object. The 'SIM' button on the dialogue box leads to the definition of parameters necessary to carry out DFD simulation. In this example, exponential distribution of 4 minutes inter-arrival rate is used to describe the arrival of TV sets; uniform distribution is used for the inspection and repair.

The simulation run parameter dialogue box is invoked from the 'Simulation' menu bar. This example looks at an 8-hour working day (480 minutes) with 120 minutes warm-up periods and 10 runs.

Note: Theoretically, the warm-up period should be determined by pilot runs of the specific simulation model. This example simply assumes that warm-up period is 1/4 of the run length.



4) HAT DFD model descriptions:

DFD Name: TV inspection DFD
Simulation parameter: Run, 10, 120, 480

DFD Node Descriptions:

Label: TVSrc Type: External Entity
A TV set arrives from the production workshop every 4 minutes. The arrival flow is continuous throughout an 8-hour working day.
Simulation parameters:
Distribution: InterArrival, Exponential, 4;
Start/end time:0.0, 480;
Assignment: inspect, 1.0;

Label: TVDest Type: External Entity
Receives tested/repaired TV sets.
Simulation parameters:

Label: inspect Type: Process
The inspection needs an inspector to check TV sets. An inspection takes 2-3 minutes. There may be 15% defect rate.
Simulation parameters:
Distribution: inspectTime, Uniform, 2, 3;
Resource: inspector;
Assignment: Repair, 0.15;TVDest, 0.85;

Label: Repair Type: Process
Repair defect TV sets. A repairman takes 15-30 minutes to repair a TV set. The repaired TV sets are sent to the packaging workshop.
Simulation parameters:
Distribution: repairTime, Uniform, 15, 30;
Resource: repairman;
Assignment: TVDest, 1.0;

DataFlow Descriptions:

Label: TV arrival Type: DataFlow
Source: TVSrc Destination: inspect

Label: Good TV Type: DataFlow
Source: inspect Destination: TVDest

Label: Defect TV Type: DataFlow
Source: inspect Destination: Repair

Label: Repaired TV Type: DataFlow
Source: Repair Destination: TVDest

5) The simulation model:

```
// context level DFD Simulation Model
Run:;
{ 10.00; 120.00; 480.00; }

Distribution:;
{
Exponential, interArrival, 4.0;
Uniform, inspectTime, 2, 3;
Uniform, repairTime, 15, 30;
}

Resource:;
{
PRIORITY, inspector;
PRIORITY, repairMan;
}

Source:;
{
TRANSACTION, DET, TVSrc_SRC, interArrival, 0.00, 480.00;
}

Queue:;
{
FIFO, inspect_Q, inspector;
FIFO, Repair_Q, repairMan;
}

Activity:;
{
inspect_Q, inspector, PROB, inspect_ACT, inspectTime;
Repair_Q, repairMan, DET, Repair_ACT, repairTime;
}

Sink:;
{
TVDest_SINK;
}

Branch:;
{
TVSrc_SRC, inspect_Q;
inspect_ACT, TVDest_SINK, 0.850;
inspect_ACT, Repair_Q, 0.150;
Repair_ACT, TVDest_SINK;
}
```

6) Simulation result and discussions:

The simulation script is forwarded to the YANSL simulation subsystem. The simulation result shows that the waiting time in the repair queue is much longer than the inspection queue (50.7 min. vs. 2.1 min.). The repairman is much busier than the inspector (89.6% vs. 52.8%). The dynamic information based on the assumption of data flow distributions and processing patterns has gone beyond the traditional DFD analysis. It suggests that the groups of inspectors and repairmen should be adjusted to balance the workload.

Unweighted Statistics

Number	Name	Min	Max	Mean	Std. Dev	# of OBS
1	inspect_Q:TIQ	0.000	18.371	2.143	1.194	1214.000
2	Repair_Q:TIQ	0.000	184.903	50.674	32.023	208.000
3	TVDest_SINK:TIS	2.000	219.443	17.122	8.154	1214.000

Time Weighted Statistics

Number	Name	Min	Max	Mean	Std. Dev	Time of OBS
1	inspector:UTL	0.463	0.630	0.528	0.203	4800.000
2	repairMan:UTL	0.599	0.962	0.896	0.349	4800.000
3	inspect_Q:ANIQ	0.000	8.000	0.454	0.292	4800.000
4	Repair_Q:ANIQ	0.000	10.000	1.839	1.381	4800.000
5	inspect_ACT:UTL	0.463	0.630	0.528	0.064	4800.000
6	Repair_ACT:UTL	0.599	0.962	0.896	0.125	4800.000

One of the scenarios is to let the inspector help the repairman to repair TV sets. As a result, the inspection becomes slower and the repair is faster. The following is the simulation output when inspection time increased to 5-10 minutes and repair time reduced to 10-20 minutes. The result shows that the utilization of the resources is more balanced (75.1% of repairman and 77.3% of inspector). The overall system performance is also better (time in system of a TV set reduced to 16.1 min. from 17.1 min.). There are more TV sets going through the system than before (1130 TV sets in the new configuration vs. 1214 TV sets before the change).

Unweighted Statistics

Number	Name	Min	Max	Mean	Std. Dev	# of OBS
1	inspect_Q:TIQ	0.000	68.740	9.496	7.542	1130.000
2	Repair_Q:TIQ	0.000	89.845	25.647	11.377	199.000
3	TVDest_SINK:TIS	3.000	157.800	16.122	7.609	1130.000

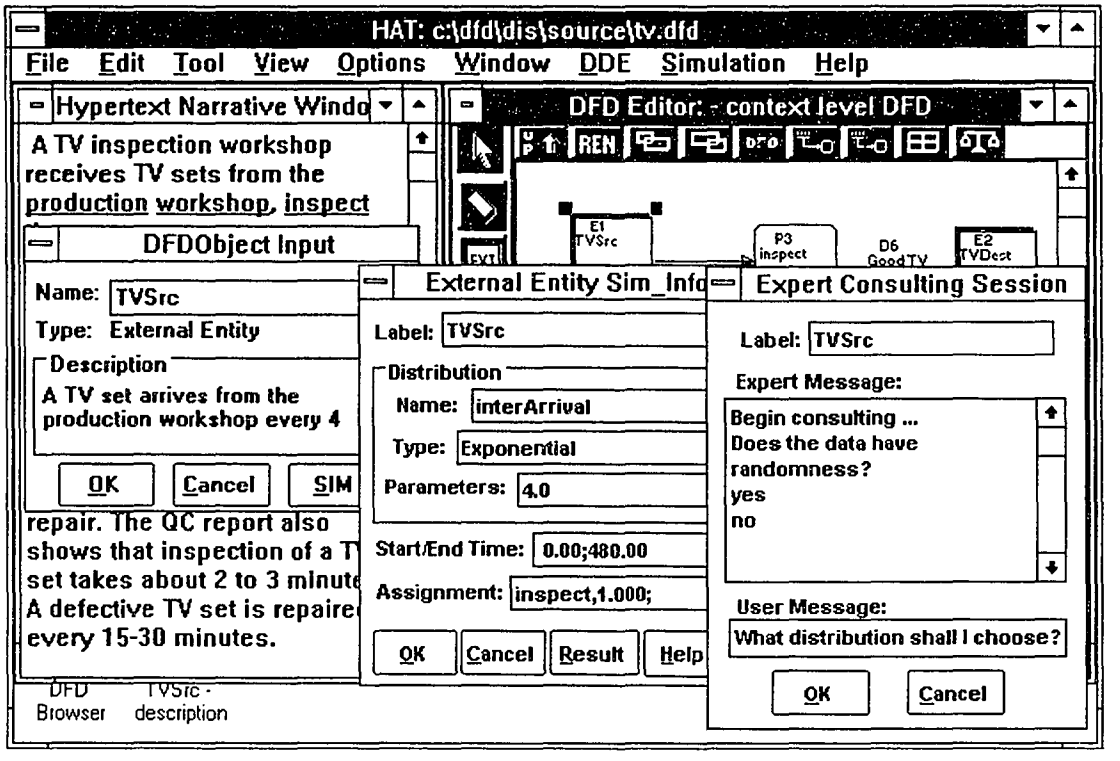
Time Weighted Statistics

Number	Name	Min	Max	Mean	Std. Dev	Time of OBS
1	inspector:UTL	0.665	0.932	0.773	0.283	4800.000
2	repairMan:UTL	0.538	0.938	0.751	0.429	4800.000
3	inspect_Q:ANIQ	0.000	20.000	2.280	2.074	4800.000
4	Repair_Q:ANIQ	0.000	5.000	1.058	0.845	4800.000
5	inspect_ACT:UTL	0.665	0.932	0.773	0.283	4800.000
6	Repair_ACT:UTL	0.538	0.938	0.751	0.429	4800.000

This example shows that HAT can help users test different scenarios and assumptions of the system workflow and understand the intrinsic dynamics of a DFD model.

7) Invoking the simulation help system

The simulation help system can be invoked at two different points: defining the simulation model and explaining the result. The 'Help' button inside simulation parameter dialogue box invokes the M4 expert system via DDE and the consulting conversation begins.



The above snapshot shows that the TVSrc DFD object dialogue box was triggered (the left) first, then a simulation parameter dialogue box (the middle) was invoked by pressing 'SIM' button, and simulation help conversation started (the right) by pressing 'Help' button.

HAT does not claim to have a sophisticated expert system built-in. However, this example does show that the conversation links between the user interface and expert system have been created. A user may consult with the expert system to determine how to choose simulation parameters and interpret simulation result.

Note: The expert system is only an add-on feature of HAT. It does not have direct impact on simulation.

8) DDE conversation log

a) A simulation task

UI: Simulation model Script....
UI: I am UI
UI: The model script is ready
SIM: Simulation finished. The result is ready.
SIM: Simulation Result ...

b) A expert system consulting session

UI: I am UI
UI:knowledge:distr.kb
UI: Data Ready
ES: Data Ready
ES:Begin consulting ...
Does the data have randomness?
yes
no

UI:yes
UI: Data Ready
ES: Data Ready
ES:Does the data have boundary?
yes
no

UI:no
UI: Data Ready
ES: Data Ready
ES:Does the data have a constant rate?
yes
no

UI:yes
UI: Data Ready
ES: Data Ready
ES:What is the time interval between two transactions?
a number.

UI:4.0
UI: Data Ready
ES: Data Ready
ES: distribution = Exponential distribution is recommended.
(100%) because kb-15.
parl = 4.0 (100%) because kb-16.
par2 = 0 (100%) because kb-17.

Consulting finished.

B.2. Analysis of an investment company: determine the system bottleneck

1) Problem description:

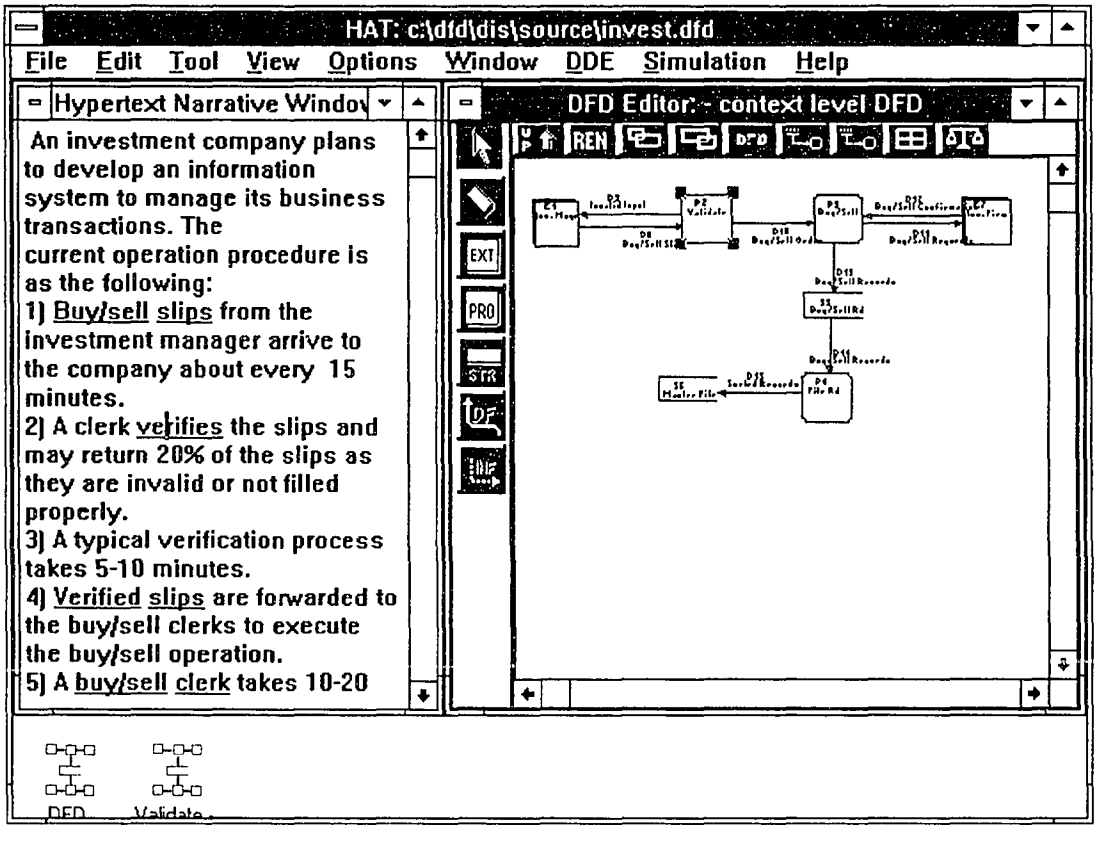
An investment company plans to develop an information system to manage its business transactions. The current operation procedure is as the following:

a) Buy/sell slips from the investment manager arrive to the company about every 15 minutes.
b) A clerk verifies the slips and may return 20% of the slips as they are invalid or not filled properly.
c) A typical verification process takes 5-10 minutes.
d) Verified slips are forwarded to the buy/sell clerks to execute the buy/sell operation.
e) A buy/sell clerk takes 10-20 minutes to file a transaction record and talk with investment firms.
f) Investment firms reply to the buy/sell clerks about every 30 minutes.
g) Transaction records have to be sorted and stored to a master file, which takes 3-5 minutes for each record.

Use a DFD model to describe the necessary components in the investment information system and find out the bottleneck in the system that should pay special attention upon implementation.

2) DFD model and hyperlinks:

Based on the problem description, a DFD model is created. The following diagram shows a snapshot of HAT interface that describes the investment firm.



3) DFD model descriptions:

DFD Name: DFD of an investment firm
Simulation parameter: Run, 10, 120, 480

DFD Node Descriptions:

Label: Inv. Mngr Type: External Entity
Investment managers send buy/sell slips at about every 15 minutes.
Simulation parameters:
Distribution: Ext1_DIS, Exponential, 15;
Start/end time: 0.0, 480;
Assignment: Validate, 1.0;

Label: Validate Type: Process
Validate buy/sell slips from investment managers. 20% of the slips may be returned to the investment managers; 80% of the slips are forwarded to buy/sell operations.
Simulation parameters:
Distribution: Pro2_DIS, Uniform, 5, 10
Resource:
Assignment: Inv. Mngr, 0.20;Buy/Sell,0.80

Label: Buy/Sell Type: Process
Perform buy/sell operations. 10-20 minutes for each transaction.
Simulation parameters:
Distribution: Pro3_DIS, Uniform, 10, 20;
Resource: Buy/Sell Rd;
Assignment: Buy/Sell Rd, 0.50;Inv. Firm, 0.50;

Label: File Rd Type: Process
Sort and store transaction records into master file. 3-5 for each record.
Simulation parameters:
Distribution: Pro3_DIS, Uniform, 3, 5;
Resource: Master File, Buy/Sell Rd;
Assignment: Master File, 1.0;

Label: Buy/Sell Rd Type: DataStore
Storage for transaction record.
Simulation parameters:
Assignment: File Rd, 1.0;

Label: Master File Type: DataStore
Master storage for all transactions.
Simulation parameters:

Label: Inv. Firm Type: ExternalEntity
Receives buy/sell requests and reply at about every 30 minutes.

Simulation parameters:
Distribution: Ext7_DIS, Exponential, 30;
Start/end time: 30, 480;
Assignment: Buy/Sell, 1.0;

DataFlow Descriptions:

Label: Buy/Sell Slips Type: DataFlow
Source: Inv. Mngr Destination: Validate

Label: Invalid Input Type: DataFlow
Source: Validate Destination: Inv. Mngr

Label: Buy/Sell Order Type: DataFlow
Source: Validate Destination: Buy/Sell

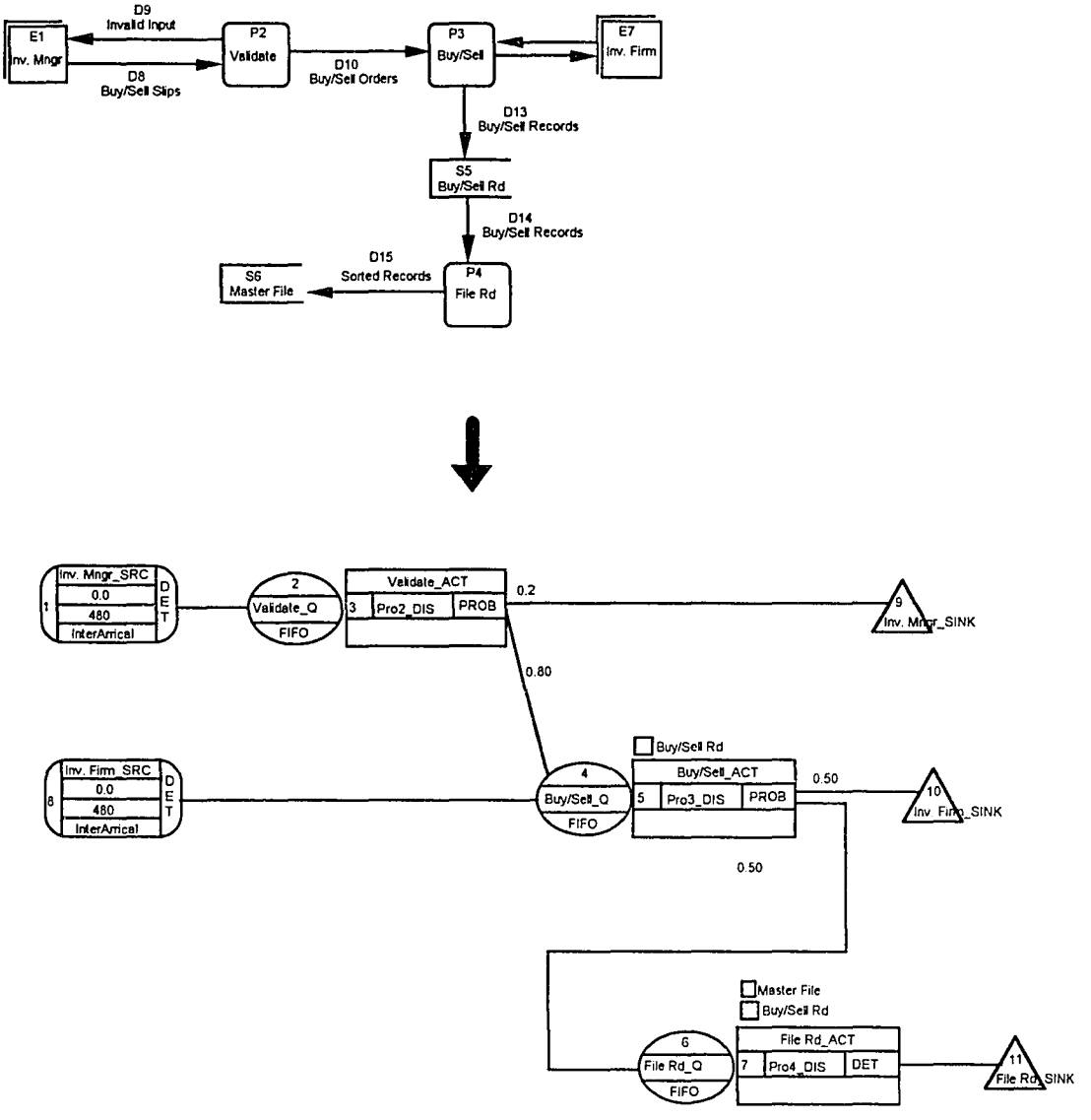
Label: Buy/Sell Requests Type: DataFlow
Source: Buy/Sell Destination: Inv. Firm

Label: Buy/Sell Confirmation Type: DataFlow
Source: Inv. Firm Destination: Buy/Sell

Label: Buy/Sell Records Type: DataFlow
Source: Buy/Sell Destination: Buy/Sell Rd

Label: Buy/Sell Records Type: DataFlow
Source: Buy/Sell Rd Destination: File Rd

Label: Sorted Records Type: DataFlow
Source: File Rd Destination: Master File



DFD - Simulation model conversion: an example of an investment company

4) The Simulation model:

```
// context level DFD Simulation Model
Run:;
{ 10.00; 120.00; 480.00; }
Distribution:;
{
Exponential, Ext1_DIS, 15;
Uniform, Pro2_DIS, 5,10;
Uniform, Pro3_DIS, 10,20;
Uniform, Pro4_DIS, 2,3;
Exponential, Ext7_DIS, 30;
}
Resource:;
{
PRIORITY, Buy/Sell Rd;
PRIORITY, Master File;
}
Source:;
{
TRANSACTION, DET, Inv. Mngr_SRC, Ext1_DIS, 0.00, 480.00;
TRANSACTION, DET, Inv. Firm_SRC, Ext7_DIS, 30.00, 480.00;
}
Queue:;
{
FIFO, Validate_Q, NULL;
FIFO, Buy/Sell_Q, Buy/Sell Rd;
FIFO, File Rd_Q, Master File;
}
Activity:;
{
Validate_Q, NULL, PROB, Validate_ACT, Pro2_DIS;
Buy/Sell_Q, Buy/Sell Rd, PROB, Buy/Sell_ACT, Pro3_DIS;
File Rd_Q, (Master File, Buy/Sell Rd), DET, File Rd_ACT, Pro4_DIS;
}
Sink:;
{
Inv. Mngr_SINK;
File Rd_SINK;
Inv. Firm_SINK;
}
Branch:;
{
Inv. Mngr_SRC, Validate_Q;
Validate_ACT, Inv. Mngr_SINK, 0.200;
Validate_ACT, Buy/Sell_Q, 0.800;
Buy/Sell_ACT, Inv. Firm_SINK, 0.500;
Buy/Sell_ACT, File Rd_Q, 0.500;
File Rd_ACT, File Rd_SINK;
Inv. Firm_SRC, Buy/Sell_Q;
}
```

5) Simulation result and discussion:

The simulation result of 10 runs shows that the Validate and File Rd queues have no delay at all. Since there is no resource requirement and process capacity limit in this model, transactions to the Validate queue should have no delay. However, it is worth noticing that the Validate activity does have a maximum number of 6 transactions in the activity at one time, which means that delay is possible if there is a resource restriction to the Validate activity. In any event, the delay will not be significant comparing with the total number of transactions going through the activity.

The biggest congestion in the system is obviously Buy/Sell queue. It has average waiting time of 86.147 minutes while other queues in the system do not have any delays. There are also a lot of transactions left over in the Buy/Sell queue when the simulation finished: 413 transactions go in the queue and only 296 (149 + 147) transactions go out the queue in 10 simulation runs. That indicates that 28.3% of the transactions are not finished in the same day. The busy rate of the Buy/Sell activity also echo the same fact that the Buy/Sell process is very busy (86% busy rate) and Buy/Sell record is the even busier (98.2% busy rate). The simulation shows that this system is not very well functioned in current situation.

Based on the simulation result, we can see that the key issue of this information system development is to focus on the Buy/Sell process and the Buy/Sell data store. The other two processes are less critical.

Unweighted Statistics

Number	Name	Min	Max	Mean	Std. Dev	# of OBS
1	Validate_Q:TIQ	0.000	0.000	0.000	0.000	309.000
2	Buy/Sell_Q:TIQ	0.000	290.730	86.147	32.333	413.000
3	File Rd_Q:TIQ	0.000	0.000	0.000	0.000	149.000
4	Inv. Mngr_SINK:TIS	5.068	9.947	7.271	0.576	66.000
5	File Rd_SINK:TIS	10.610	263.830	104.166	29.571	127.000
6	Inv. Firm_SINK:TIS	18.300	300.200	105.736	7.609	147.000

Time Weighted Statistics

Number	Name	Min	Max	Mean	Std. Dev	Time of OBS
1	Buy/Sell Rd:UTL	0.977	0.989	0.982	0.010	4800.000
2	Master File:UTL	0.109	0.152	0.121	0.030	4800.000
3	Validate_Q:ANIQ	0.000	0.000	0.000	0.000	4800.000
4	Buy/Sell_Q:ANIQ	0.000	26.000	7.728	3.890	4800.000
5	File Rd_Q:ANIQ	0.000	0.000	0.000	0.000	4800.000
6	Validate_ACT:UTL	0.000	6.000	0.482	0.131	4800.000
7	Buy/Sell_ACT:UTL	0.000	1.000	0.860	0.131	4800.000
8	File Rd_ACT:UTL	0.000	1.000	0.121	0.124	4800.000

APPENDIX C. HAT SURVEY QUESTIONNAIRE

HAT Questionnaire

I. Circle the number that indicates your level of agreement with each statement-

1. HAT was very helpful in understanding process and data modeling?

|_____|_____|_____|_____|
1 2 3 4 5

disagree agree completely

2. HAT was easy to use

|_____|_____|_____|_____|
1 2 3 4 5

disagree agree completely

3. There was NO relationship between the process and data modeling concepts in the textbook and HAT helped to bridge the gap.

|_____|_____|_____|_____|
1 2 3 4 5

disagree agree completely

II. Please respond to the following questions by circling the most appropriate answer

4. How long did it take to learn the basic functions of HAT (the DFD editor, the ERD editor, creating hyperlinks and browsing DFDs)?

(a) Less than 2 hours.
(b) 2-5 hours.
(c) One day.
(d) More than one day.

5. Do you think that the use of hypertext links between text in the narrative window and graphical objects was helpful for learning about process and data modeling?

(a) Very helpful.
(b) Helpful.
(c) Confusing.
(d) The hyperlinks had not effect.

6. How do you like the multiple-window user interface?

(a) Easy to use, just like other Windows applications.
(b) I can manage it, but I'm not happy with the way it looks.
(c) I am totally lost in the windows.

7. How do you like the graphical tools?

(a) I like the drag and drop features as well as the tool bars and control bars.
(b) The tools are useful, but could be designed better.
(c) I don't feel comfortable with the 'look' and 'feel' of the tools.

8. How do you like the navigation through DFDs via button clicks?

(a) Very easy to navigate among DFDs and helpful in understanding functional decomposition.
(b) Helpful, but not easy to use.
(c) Confusing, not helpful at all.

9. Was it easy to create a hyperwords and hyper links?
   (a) Easy to use.
   (b) I can learn it, but somewhat confusing.
   (c) Hard to learn.

10. Does the HELP facility really help?
   (a) Very helpful.
   (b) OK, but it needs some work.
   (c) Not helpful.

11. Is the reporting and printing function sufficient for your homework?
   (a) I like it.
   (b) I can live with it, but more work needs to be done on the printing function.
   (c) It is terrible.

12. Is the response time for operations in HAT satisfactory?
   (a) Yes (b) No

13. Where do you use HAT?
   (a) In the student lab (b) At home c) Both (a) and (b)

14. Do you have computer at home?
   (a) Yes (b) No
   if yes, can HAT run on your home computer?
   (a) Yes (b) No

15. Have you used any Windows applications before?
   (a) Yes (b) No

III. Please answer each question as indicated.

16. Assuming that HAT were a bug-free commercial product, what would be a fair price for it? $_____

17. What type of computer do you have? CPU, RAM, hard drive size, screen size, color/bw?

_____

Please place additional comments about HAT on the lines below-

_____
_____
_____
_____
_____
_____
_____
_____

Thank you very much for taking this survey

APPENDIX D. M4 RULE-BASE EXAMPLES IN HAT

1) Determine a distribution

```
/* Goal */
goal = [distribution, par1, par2].

/* Facts */
distr(no_distribution) = ' No distribution is recommended.'.
distr(uniform) = 'Uniform distribution is recommended.'.
distr(normal) = 'Normal distribution is recommended.'.
distr(exponential) = 'Exponential distribution is recommended.'.
p1(no_random) = 0.
p2(no_random) = 0.
prepend(X) = float(1/X).


/* Rules */

 /* no_random */
if random=no and distr(no_distribution) = N and par1=P1 and par2 = P2
 then distribution = N.
if  random = no and p1(no_random) =P
 then par1 = P.
if  random = no and p2(no_random) =P
 then par2 = P.


  /* uniform */
if random  and bounded and distr(uniform) = N and par1 = P1 and par2 = P2
  then distribution = N.
if random and bounded and lower  = D
  then par1 = D.
if random and bounded and upper = D
  then  par2 = D.

 /* exponential */
if random and bounded = no and const_rate and distr(exponential) = N and
par1 = P1 and par2 = P2
  then distribution = N.
if random and bounded = no and const_rate and num = D and 1/D = P
  then par1 = P.
if random and bounded = no and const_rate
  then  par2 = 0.

 /* normal */
if random and bounded = no and const_rate=no and distr(normal) = N and
par1=P1 and par2 = P2
  then distribution = N.
if random and bounded = no and const_rate = no and  avg = D
  then par1 = D.
```

```
if random and bounded = no and const_rate = no and var = D
  then  par2 = D.


/* Questions */
question(random) = 'Does the data have randomness?'.
legalvals(random) = [yes, no].

question(bounded) = 'Does the data have boundary?'.
legalvals(bounded) = [yes, no].

question(const_rate) = 'Does the data have a constant rate?'.
legalvals(const_rate) = [yes, no].

question(upper) = 'What is the upper boundary?'.
legalvals(upper) = number.

question(lower) = 'What is the lower boundary?'.
legalvals(lower) = number.

question(num) = 'How many evets happen in a unit time?'.
legalvals(num) = number.

question(avg) = 'What is the average of the data?'.
legalvals(avg) = number.

question(var) = 'What is the range of change?'.
legalvals(num) = number.
```

2) Determine the runs, length, and warmup period

```
/* Goal */
goal = [run, length, warmup].

/* Facts  (default values)*/
rdef = 10.
ldef = 480.

/* Rules */
  /* use default_val */
if def = yes and rdef = P
  then r = P.

if def = yes and ldef = P
  then length = P.

if def = no and  r = P and rdef = P1 and P1 > P
 then r = P1.

if def = no and l = P and ldef = P1 and P1>P
  then length = P1.
```

```
if length = P and P/4 = W
 then warmup = W.

 /* use user input value */
if def = no and r = P and rdef = P1 and P>P1
 then r = P.

if def = no and l= P and ldef = P1 and P > P1
  then length = P.

 /* conclusion */
if r = P and length = P1 and warmup = P2
 then run = P.

/* Questions */
question(r) = 'How many runs?'.
legalvals(r) = integer.

question(l) = 'How long is each run?'.
legalvals(l) = integer.

question(def) = 'Do you want to use default value?'.
legalvals(def) = [yes, no].
```

3) Simulation result interpretation

```
/* Goal */
goal = [recommendation].

/* Facts */
recom(time_in_Q, low) = 'Low time in Q. All transactions are processed promptly with little
delay. This node is not a bottleneck in the system.'.
recom(time_in_Q, normal) = 'Time in Q is normal.'.
recom(time_in_Q, high) = 'Time in Q is too long, possible bottleneck.'.
recom(time_in_Q, highmax) = 'Time in Q is normal, but maximum value is too high that
indecates short congestions.'.

recom(num_in_Q, low) = 'Low number in Q. All transactions are processed promptly with little
delay. This node is not a bottleneck in the system.'.
recom(num_in_Q, normal) = 'Number in Q is normal.'.
recom(num_in_Q, high) = 'Number in Q is too high, possible bottleneck.'.
recom(num_in_Q, highmax) = 'Number in Q is normal, but maximum value is too high that
indecates short congestions.'.

recom(time_in_system, low) = 'Low time in system. All transactions are processed promptly
with little delay. The system has not a bottleneck.'.
recom(time_in_system, normal) = 'Time in system is normal.'.
recom(time_in_system, high) = 'Time in system is too long, possible bottleneck in the system.'.
recom(time_in_system, highmax) = 'Time in system is normal, but maximum value is too high
that indecates short congestion.'.
```

```
recom(resr_utl, low) = 'Low resource utilization. The resource is idle most of the time.
Redundant or under performed resources.'.
recom(resr_utl, normal) = 'Resource utilization is normal.'.
recom(resr_utl, high) = 'Resource utilization is too high, possible bottleneck.'.
recom(resr_utl, highmax) = 'Resource utilixation is normal, but maximum value is too high
that indecates short congestions.'.

recom(act_utl, low) = 'Low activity utilization. The activity is idle most of the time. Redundant
or under performed activity.'.
recom(act_utl, normal) = 'Activity utilization is normal.'.
recom(act_utl, high) = 'Activity utilization is too high, possible bottleneck.'.
recom(act_utl, highmax) = 'Activity utilization is normal, but maximum value is too high that
indecates short congestions.'.

recom(norecom) = 'Sorry, I cannot provide any recommendation on this issue'.

/* Rules */
/* time_in_Q */
if type = time_in_Q and length = P and mean = P1 and max = P2
  and P1< P/10 and recom(time_in_Q,low) = R
  then recommendation = R.
if type = time_in_Q and length = P and mean = P1 and max = P2
  and P1> P/10 and P1< P/4 and P2>P/4 and recom(time_in_Q,highmax) = R
  then recommendation = R.
if type = time_in_Q and length = P and mean = P1 and max = P2
  and P1> P/10 and P1< P/4 and P2<P/4 and recom(time_in_Q,normal) = R
  then recommendation = R.
if type = time_in_Q and length = P and mean = P1 and max = P2
  and P1> P/4 and recom(time_in_Q,high) = R
  then recommendation = R.

/* time_in_system */
if type = time_in_system and length = P and mean = P1 and max = P2
  and P1< P/5 and recom(time_in_system,low) = R
  then recommendation = R.
if type = time_in_system and length = P and mean = P1 and max = P2
  and P1> P/5 and P1< P/2 and P2>P/2 and recom(time_in_system,highmax) = R
  then recommendation = R.
if type = time_in_system and length = P and mean = P1 and max = P2
  and P1> P/5 and P1< P/2 and P2<P/2 and recom(time_in_system,normal) = R
  then recommendation = R.
if type = time_in_Q and length = P and mean = P1 and max = P2
  and P1> P/2 and recom(time_in_system,high) = R
  then recommendation = R.

/* resr_utl */
if type = resr_utl and mean = P1 and max = P2
  and P1< 0.3 and P2 < 0.8 and recom(resr_utl,low) = R
  then recommendation = R.
if type = resr_utl and mean = P1 and max = P2
  and P1> 0.3 and P1< 0.8 and P2>0.8 and recom(resr_utl,highmax) = R
```

```
 then recommendation = R.
if type = resr_utl and mean = P1 and max = P2
 and P1> 0.3 and P1< 0.8 and P2<0.8 and recom(resr_utl,normal) = R
 then recommendation = R.

if type = resr_utl and mean = P1 and max = P2
 and P1> 0.8 and recom(resr_utl,high) = R
 then recommendation = R.

/* act_utl */
if type = act_utl and mean = P1 and max = P2
 and P1< 0.3 and P2 < 0.8 and recom(act_utl,low) = R
 then recommendation = R.
if type = act_utl and mean = P1 and max = P2
 and P1> 0.3 and P1< 0.8 and P2>0.8 and recom(act_utl,highmax) = R
 then recommendation = R.
if type = act_utl and mean = P1 and max = P2
 and P1> 0.3 and P1< 0.8 and P2<0.8 and recom(act,normal) = R
 then recommendation = R.
if type = act_utl and mean = P1 and max = P2
 and P1> 0.8 and recom(act_utl,high) = R
 then recommendation = R.

/* no recommendation */
if recom(norecom) = P
 then recommendation = P.

/* Questions */
question(type) = 'What type of statistics has been collected?'.
legalvals(type) = [time_in_Q, num_in_Q, time_in_system, resr_utl, act_utl].

question(run) = 'How many runs were tried?'.
legalvals(run) = integer.

question(length) = 'How long was each run?'.
legalvals(length) = integer.

question(max) = 'What is the maximum value?'.
legalvals(max) = number.

question(mean) = 'What is the average value?'.
legalvals(mean) = number.

question(std) = 'What is the standard deviation?'.
legalvals(std) = number.

/* get more information of connections in complicated cases*/
```

APPENDIX E. SELECTIVE CLASS DESCRIPTIONS OF HAT

E.1. The user interface

1) The MDI base window

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (HAT) (R)  Main Window
// MDI Base Window for all other windows
// Handling messages from its children and scheduling for different tasks.

//        superclass:  MyDDEWindow
//        File:      dfdwin.h
//        Author: Jackson He
//        Date:    03/93
//        Language:C++
//        Modification notes:

#ifndef __DFDWIN_H
#define __DEFWIN_H

#include "define.h"
#include "objdraw.h"
#include "mydde.h"

#define CM_ArrangeAll   204
#define CM_Parameter    205
#define CM_RunSim       206
#define CM_ShowDDELog           207
#define D_ABOUT         111
#define NumDataWin    30
#define SCRIPTLEN       3000
_CLASSDEF(TEditWindow)

//****  Declare TDFDApp, starting point of the program ******//
class _CLASSTYPE TDFDApp : public TApplication
{
public:
        TDFDApp(LPSTR name, HINSTANCE hInstance,
                  HINSTANCE hPrevInstance, LPSTR lpCmd,  int nCmdShow)
                : TApplication(name, hInstance, hPrevInstance, lpCmd, nCmdShow){};
        virtual void InitMainWindow();
        virtual void InitInstance();
};  // end of TDFDApp

// **** Declare TDFDWindow, the main MDI window *****//
class _CLASSTYPE TDFDWindow : public MyDDEWindow
{
        friend UIAppHandle;
private:
        Boolean Lock;
        virtual void SetupWindow(); // window setup
```

```
TEditWindow* ShowTextWin(char* text, char* title);
 Boolean TimeBomb(int day, int year);
 virtual BOOL CanClose();
 void LockMe();
 void UnlockMe();
int IsActivate(RTMessage Msg);
void MenuEnable();
void MenuEnableND();
        // enable menu for non-draw window

// message handler
virtual void DFDError(RTMessage Msg) = [WM_FIRST + WM_DFDError];
        // Report DFD errors
virtual void DataWinClosed(RTMessage Msg) = [WM_FIRST + WM_DataWinClosed];
        // Msg when a DataWindow is closed
virtual void WMSize( TMessage& Message ) = [ WM_FIRST + WM_SIZE ];
        // Resize window
virtual void ArrangeAll(RTMessage Msg) = [CM_FIRST + CM_ArrangeAll];
        // arrange all children
virtual void CMShowLog(RTMessage Msg) = [CM_FIRST + CM_ShowDDELog]
   { ShowTextWin((char*)ConvLog.data(), "DDE Conversation Log");} // show DDE log
virtual void CMAbout(RTMessage Msg) = [CM_FIRST + CM_ABOUT];  // about msg
virtual void CMFloating(RTMessage Msg) = [CM_FIRST + CM_FLOATING];
         // turn on/off floating label
virtual void OpenFile(RTMessage Msg) = [CM_FIRST + CM_FILEOPEN];
         // open file operatio
virtual void NewFile(RTMessage Msg) = [CM_FIRST + CM_FILENEW];
        // new file operation
virtual void CMFileSave(RTMessage Msg) = [CM_FIRST + CM_FILESAVE];
        // save file operation
virtual void CMFileSaveAs(RTMessage Msg) = [CM_FIRST + CM_FILESAVEAS];
         // saveas opertation
virtual void CMPrintProject (RTMessage Msg) = [CM_FIRST + CM_PRINTPROJECT];
        // print project
virtual void CMShowPDict(RTMessage Msg) = [CM_FIRST + CM_ShowPDict];
         // display ProjectDictionary
 virtual void CMShowDDict(RTMessage Msg) = [CM_FIRST + CM_ShowDDict];
        //display DataDictionary
 virtual void WMShowERD(RTMessage Msg) = [WM_FIRST + WM_ShowERD];
         // show ERD
 virtual void WMShowDDWin(RTMessage Msg) = [WM_FIRST + WM_ShowDWin];
        // show DataWindow
virtual void RenewDFD(RTMessage Msg) = [WM_FIRST + WM_RenewList];
         // reset all objects when a new project is started
virtual void ShowDFD(RTMessage Msg) = [WM_FIRST + WM_ShowDFD];
        // msg when select a new DFD from broswer
virtual void ShowDFDObject(RTMessage Msg) = [WM_FIRST + WM_ShowDFDObject];
        //show dfdobjects set by DataWinddow or project dictionary
 virtual void ShowMsgWin(RTMessage Msg) = [WM_FIRST + WM_ShowMsgWin];
        // show message window after checking DFD
virtual void MarkNode(RTMessage Msg) = [WM_FIRST + WM_MarkNode];
        // Highlight current DFD object, triggered by a msg
```

```
        virtual void SetIndex(RTMessage Msg) = [WM_FIRST + WM_SetIndex];
                // Set index in the DFD browser.  msg when slect an object in DFDdraw window
        virtual void ReExpand(RTMessage Msg) = [WM_FIRST + WM_ReExpandList];
                // msg when DFDdraw updates nodes
        virtual void ShowScript(RTMessage Msg) = [WM_FIRST + WM_ShowScript];
                // msg when description is shown with control button
         virtual void UpdateDDict(RTMessage Msg) = [WM_FIRST + WM_UpdateDDict];
                // update data dictionary
        virtual void WMHyperWordSelected(RTMessage Msg) =
                [WM_FIRST +WM_HyperWordSelect];  // msg when a hyperword is chosen
        virtual void WMRedrawDFD(RTMessage Msg) = [WM_FIRST + WM_RedrawDFD];
                // redraw current DFD
        virtual void WMSetDataFocus(RTMessage Msg) = [WM_FIRST + WM_SetDataFocus];
                // set focus to a DataWindow
        virtual void CMParameter(RTMessage Msg) = [CM_FIRST + CM_Parameter];
         virtual void CMSimRun(RTMessage Msg) = [CM_FIRST + CM_RunSim];
                // handlers to the simulation menu
        virtual void CMHelp(RTMessage Msg) = [CM_FIRST + CM_HELP];  // Help

        // simulation methods
        void SetRunParameter();
         DFDSet* CheckSimParameter(DFDSet[]);
         void PollSimParameter(); //poll every node in current DFD for simulation parameters
         Boolean GenSimScript(); // Generate simulation script from DFD model
         // simulation result parser
         void ParseResult(char result[]);
         int GetNextLine(char buffer[], char temp[], int start);
         int GetItems(char Line[], int from, int to, char temp[]);

public:
        WORD ChildNum;
        PTDFDDrawWindow DFDDraw;
        PBrowserWindow Browser;
        PTDescriptWindow Descript;
        PpDictWindow pDict;
        PdDictWindow dDict;
        PTERDDrawWindow ERDWin;
        PMessageWindow MsgWin;
        DataWindow** DataWinHandle[NumDataWin];
        int DataWinCount;
        PHyperTextWindow HTxtWin;
        char ScriptBuffer[SCRIPTLEN];

         TDFDWindow(LPSTR ATitle, LPSTR MenuName);
        ~TDFDWindow();
        UIAppHandle* GetAppHandle(){return(UIAppHandle*)Appl;}
        virtual void CloseAuxWin(); // close auxilary windows, such as Dictionary Window

        // misc methods
        virtual void MarkNode(DFDObject* obj); //Highlight a given DFD Object
        virtual void PrintText(char* text, int len);
        virtual void RedrawDFD(DFD* dfd);
```

```
        virtual void CMHWordConnect(); // connect current hyperword and a dfd object
        virtual void CMHWordDisconnect();  // disconnect current hyperword and a dfd object
        virtual void CMERDHWConnect();  // connect current hyperword and an ERD object
        virtual void CMERDHWDisconnect();  // disconnect current hyperword and an ERD object

        void GetWindowClass( WNDCLASS& WndClass );
        void HighlightDFDWin(RWCollectable* data, RTMessage Msg);
         void HighlightEntity(RWCollectable* data, RTMessage Msg);
        void HighlightRelation(RWCollectable* data, RTMessage Msg);
                // highlight object in DFD or ERD window once a data pointer
                // is selected from the hypertext window
};
#endif // end of TDFDWindow
```

2) The hypertext window

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  HypertextWindow
//File: htxtwin.h
//Date: 4/29/93
//Author: Tien Lum & Jackson He

#ifndef HTXTWIN_h
#define HTXTWIN_h

#include "hfilewin.h"
#define  CM_REDRAW          860
#define  CM_PRINT           706

//******** HyperTextWindow ********//
class HyperTextWindow : public HyperFileWindow
{
  protected:
        virtual void CMFileNew(RTMessage Msg) = [CM_FIRST + CM_FILENEW];
        virtual void CMFileOpen(RTMessage msg) = [ CM_FIRST + CM_FILEOPEN ];
        virtual void CMFileSaveAs(RTMessage msg) = [ CM_FIRST + CM_FILESAVEAS ];
        virtual void CMFileSave(RTMessage msg) = [ CM_FIRST + CM_FILESAVE ];
        virtual void CMFilePrint       (RTMessage Msg) = [CM_FIRST + CM_PRINT];
        //virtual void CMPrintProject (RTMessage Msg) = [CM_FIRST + CM_PRINTPROJECT];
        virtual int IsActivate(RTMessage Msg) = [WM_FIRST + WM_CHILDACTIVATE];
        virtual void CMRedraw(RTMessage Msg) = [CM_FIRST + CM_REDRAW];
        virtual void HyperWordSelected(RTMessage msg) = [ WM_FIRST + WM_HyperWordSelect ];
        virtual void SetupWindow();
        void MenuEnable();

public:
        Boolean Close;
        HyperTextWindow(PTWindowsObject parent, LPSTR title, LPSTR fname);
        ~HyperTextWindow();
        Boolean CanClear(){return TRUE;}
        Boolean CanClose();
        HyperWord *GetHyperWord(RWCollectable* data)
```

```
        {       // return the hyperword for given user data
                return GetHyperEditor()->GetHyperWord(data);
        };
        void ResetUserData(RWCollectable* data) { GetHyperEditor()->ResetUserData(data);}
        void HighLight(RWCollectable* data);
        void NewFile();
};
#endif //HyperTextWindow
```

3) The DFD editor window

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  DFDDrawWindow
// DFDDrawWindow
// Base Window for DrawWindowDFD and DFD tools
// interface with MDI window

//      SuperClass: ObjectDrawWindow
//      File:   dfddraw.h
//      Author: Jackson He
//      Date:   03/93
//      Language:C++
//      Modification notes:

#ifndef __DFDDRAW_H
#define __DFDDRAW_H

#define __OBJDRAW_CPP
#include "objdraw.h"
_CLASSDEF(DrawWinDFD)

//********* TDFDDrawWindow ***********//
class TDFDDrawWindow : public ObjectDrawWindow
{
 protected:
        RWCString Text;
        void PrepareText(DFD* dfd); // prepare Text for printing
        void PrintDFD(DFD* dfd);// print DFD
        virtual BOOL CanClose();
        virtual void SetupWindow();
        virtual int IsActivate(RTMessage Msg) = [WM_FIRST + WM_CHILDACTIVATE];
        virtual void WMSize(RTMessage Msg) = [WM_FIRST + WM_SIZE];
        virtual int  FileSaveAs();
        virtual void Save();
        void PrintProjectTree();
        virtual void LeftButtonUp(PTPoint MousePt, WORD InputStates);
        virtual void DefCommandProc(RTMessage Msg);
        void MenuEnable();

 public:
        BOOL Close;
        PTToolBar     ToolBar;
```

```
        PTControlBar    ControlBar;
        TDFDDrawWindow(PTWindowsObject AParent, LPSTR ATitle);
        ~TDFDDrawWindow();

        // message handler
        virtual void CMFileNew(RTMessage Msg) = [CM_FIRST + CM_FILENEW];
        virtual void CMFileOpen(RTMessage Msg) = [CM_FIRST + CM_FILEOPEN];
        virtual void CMFileSave(RTMessage Msg) = [CM_FIRST + CM_FILESAVE];
        virtual void CMFileSaveAs(RTMessage Msg) = [CM_FIRST + CM_FILESAVEAS];
        virtual void CMFilePrint(RTMessage Msg) = [CM_FIRST + CM_PRINT];
        virtual void CMActualSize(RTMessage Msg) = [CM_FIRST + CM_ACTUAL];
        virtual void CMToolZoom() = [CM_FIRST + CM_ZOOM];
        virtual void PrintProject (RTMessage Msg) = [CM_FIRST + CM_PRINTPROJECT];
        virtual void SetTool(int ToolId);       // Called by DefWindProc
        virtual void SetControl(int ControlId);
        //utility
        virtual void SetCaption();
        virtual void SizeKids();
        PDrawWinDFD  GetDrawWin(){return (DrawWinDFD*) DrawWindow;}
};
#endif //TDFDDrawWindow
```

4) The DFD drawing canvas

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  DrawWindowDFD
// provides drawing canvas for DFD drawing
// basic drawing operations and connections with
// data repository are setup

//      SuperClass: TDrawWiindow
//      File:    dwindfd.h
//      Author: Jackson Hc
//      Date:    03/93
//      Language:C++
//      Modification notes:

#ifndef __DRAWWINDFD_H
#define __DRAWWINDFD_H

#ifndef __GWINDOW_H
#include "gwindow.h"
#endif

#include "global.h"
#include "drawwin.h"

_CLASSDEF(DrawWinDFD)
_CLASSDEF(VisualRepDFD)
_CLASSDEF(DFD)
```

```
// ******** DrawWinDFD **********//
class DrawWinDFD : public TDrawWindow
{
protected:
        // mouse move
        virtual void LeftButtonDown(PTPoint MousePt, WORD InputStates);
        virtual void BeginEraser(PTPoint MousePt, WORD InputStates);
        virtual void BeginDFDDraw(PTPoint MousePt, WORD InputStates, char* Id, char* label);
        virtual void Drag(PTPoint MousePt, WORD InputStates);
        virtual void LeftButtonUp(PTPoint MousePt, WORD InputStates);
        virtual void EndEraser(PTPoint MousePt, WORD InputStates);
        virtual void EndDFDDraw(PTPoint MousePt, WORD InputStates, char* Id, char* label);
        virtual void EndSelectDrag(PTPoint MousePt, WORD InputStates);

        // Window Messages
        virtual void WMRButtonDown  (RTMessage Msg) =
                [WM_FIRST + WM_RBUTTONDOWN];
        virtual void WMLButtonDblClk(RTMessage Msg) =
                [WM_FIRST + WM_LBUTTONDBLCLK];
        void WMChar(RTMessage msg) = [WM_FIRST + WM_CHAR];

        // Misc
        virtual void BindingDrag(PTPoint MousePt);
        void RedrawDFD();
        void ExplodePRO(Process* pro);
        void GenInterfaceNodes(Process* pro, DFD* subDFD);
        void GotoParent();
        void CMDeleteDFD();  // delete current DFD and move to parent;

public:
        DFD * CurDFD;
        PTGraphic   SNode, DNode;
        DrawWinDFD(PTGWindow AParent);
        ~DrawWinDFD();
        virtual BOOL SetLabel(DFDObject* obj);
        virtual void MarkFlow(VisualRepDFD* vflow);
        virtual void SetTool(int ToolID);
        virtual void MarkPicture(VisualRepDFD* apicture);
        VisualRepDFD * CreateNode(char* label, int type, TPoint * center);
        VisualRepDFD * CreateFlow(char * label);
        void CreateDFD();
        void readDFDName(char* name, int opr);
        void SetDFDName();
        void MarkNode(DFDObject* obj);
        void AddInterfaceNode(DFDNode*, DFD*, Boolean, int);
        VisualRepDFD* GetBuddyLabel(VisualRepDFD* vflow);
        VisualRepDFD* GetBuddyFlow(VisualRepDFD* vlabel);
        VisualRepDFD* GetBuddy(VisualRepDFD* vrep);
                // from a flow find its label, and vise versa
         void MarkBuddy(VisualRepDFD * vrep);
}; // end of DrawWinDFD
#endif
```

5) The ERD editor window

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  ERDDrawWindow
// Base Window for DrawWindowERD and ERD tools
// interface with MDI window
//        SuperClass: ObjectDrawWindow
//        File:     erddraw.h
//        Author: Jackson He
//        Date:     03/93
//        Language:C++
//        Modification notes:

#ifndef __ERDDRAW_H
#define __ERDDRAW_H

#include "dwinerd.h"

#define __OBJDRAW_CPP
#include "objdraw.h"

_CLASSDEF(DrawWinDFD)
_CLASSDEF(ERDCtrlBar)

// ************* TERDDrawWindow ************//
class TERDDrawWindow : public ObjectDrawWindow
{
 protected:
        RWCString Text;
        void PrepareText();
         // message handler
        virtual void LeftButtonUp(PTPoint MousePt, WORD InputStates);
        virtual void DefCommandProc(RTMessage Msg);
        virtual void CMActualSize      (RTMessage Msg) = [CM_FIRST + CM_ACTUAL];
        virtual void CMToolZoom        (RTMessage Msg) = [CM_FIRST + CM_ZOOM];
        virtual int IsActivate(RTMessage Msg) = [WM_FIRST + WM_CHILDACTIVATE];
        virtual void WMSize(RTMessage Msg) = [WM_FIRST + WM_SIZE];
         virtual void CMFilePrint(RTMessage Msg) = [CM_FIRST + CM_PRINT];
         virtual BOOL CanClose();
        virtual void SetupWindow();
 public:
        ER_Diagram* CurERD;
        PERDTool     ToolBar;
        PERDCtrlBar     ControlBar;
        Boolean Close;
        TERDDrawWindow(PTWindowsObject AParent, ER_Diagram* curerd);
        ~TERDDrawWindow();

          //tool
        virtual void SetTool(int ToolId);       // Called by DefWindProc
        virtual void SetControl(int ControlId);
```

```
        //utility
        virtual void SetCaption();
        virtual void SizeKids();
        PDrawWinERD  GetDrawWin(){return (DrawWinERD*) DrawWindow;};
        void GrabDataObject();  // Grab selected data object from data dictionay window
                                // to current ERD
        void ShowDataWindow();  // show the datawindow connected with current ERD
        void MenuEnable();
 }; //end of TERDDrawWindow
#endif
```

6) The ERD drawing canvas

```
Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  DrawWindowERD
//DrawWindowERD
// provides drawing canvas for ERD drawing
// basic drawing operations and connections with
// data repository are setup
//      SuperClass: TDrawWindow
//      File:    dwinerd.h
//      Author: Jackson He
//      Date:    03/93
//      Language:C++
//      Modification notes:

#ifndef __DRAWWINERD_H
#define __DRAWWINERD_H
#ifndef __GWINDOW_H
#include "gwindow.h"
#endif

#include "global.h"
#include "drawwin.h"

_CLASSDEF(DrawWinERD)
_CLASSDEF(VisualRepERD)
_CLASSDEF(ER_Diagram)

//******* DrawWinERD ********//
class DrawWinERD : public TDrawWindow
{
protected:
        virtual void SetupWindow()  { TDrawWindow :: SetupWindow(); }
        // mouse move
        virtual void LeftButtonDown(PTPoint MousePt, WORD InputStates);
        virtual void BeginEraser(PTPoint MousePt, WORD InputStates);
        virtual void BeginERDDraw(PTPoint MousePt, WORD InputStates);
        virtual void Drag(PTPoint MousePt, WORD InputStates);
        virtual void LeftButtonUp(PTPoint MousePt, WORD InputStates);
        virtual void EndEraser(PTPoint MousePt, WORD InputStates);
```

```
        virtual void EndERDDraw(PTPoint MousePt, WORD InputStates);
        virtual void EndSelectDrag(PTPoint MousePt, WORD InputStates);
        // Window Messages
        virtual void WMRButtonDown  (RTMessage Msg) = [WM_FIRST + WM_RBUTTONDOWN];
        virtual void WMLButtonDblClk(RTMessage Msg) =
                                        [WM_FIRST+WM_LBUTTONDBLCLK];
public:
        ER_Diagram * CurERD;
        PTGraphic   SNode, DNode;
        DrawWinERD(PTGWindow AParent, ER_Diagram* curerd);
        ~DrawWinERD();
        // Misc
        virtual void MarkRelation(VisualRepERD* vflow);
        virtual void SetTool(int ToolID);
        virtual void MarkPicture(VisualRepERD* apicture);
        VisualRepERD * CreateEntity(TPoint * mousePt);
        VisualRepERD * CreateRelation();
        void CreateERD();
        virtual void BindingDrag(PTPoint MousePt);
        void RedrawERD();
        void MarkNode(Entity* entity);
        void MarkRelation(Relation* relation);
}; // end of  DrawWinERD
#endif // __DRAWWINERD_H//
```

7) The graphical object class for DFD

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  VisualRepDFD
// inherited from VisualRep, serve as the objects
// shown in the DFD drawing window
//      SuperClass: VisualRep
//      File:     visdfd.h
//      Author: Jackson He
//      Date:    03/93
//      Language:C++
//      Modification notes:

#ifndef _VisualRepDFD
#define _VisualRepDFD

#include "visrep.h"

// ***** VisualRepDFD ********//
class VisualRepDFD : public VisualRep
{
  public:
        VisualRepDFD* Buddy; //used for label object;
        VisualRepDFD(VisualObject * interface);
        ~VisualRepDFD();
         // Basic operations
        VisualObject * GetInterface() {return(VisualObject*) Interface;};
```

```
        DFD* GetCurrentDFD();
        DFDObject *GetDFDObject();
        DataNode * GetDataNode();
        void SetLabel(char * label);
        DFDType GetType();
        RWCString GetId();
        int GetLevel();
        char* GetLabel();
        TPoint * GetCenter();
        TPoint * GetStart(){ return Start;}
         TPoint * GetEnd() { return End;}
        RWDlistCollectables* GetInFlows();
        RWDlistCollectables* GetOutFlows();
        void Hilight(VisualRepDFD* picture);
         // Draw DFD
        void ConstructNode(DFDType type);
        void ConstructFlow();
         void DrawFlow(DFDNode* source, DFDNode* dest);
}; // end of VisualRepDFD
#endif
```

8) The graphical object for ERD

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  VisualRepERD
// inherited from VisualRep, serve as the objects
// shown in the ERD drawing window
//        SuperClass: VisualRep
//        File:     viserd.h
//        Author: Jackson He
//        Date:    03/93
//        Language:C++
//        Modification notes:

#ifndef _VisualRepERD
#define _VisualRepERD

#include "visrep.h"
#include "erd.h"

// ****** VisualRepERD *******//
class VisualRepERD : public VisualRep
{
  public:
        Boolean IsEntity;
        VisualRepERD(RWCollectable * interface, Boolean isEntity);
         ~VisualRepERD();

         // Basic operations
        RWCollectable* GetInterface(){return Interface;}
```

```
        Entity *GetEntity()
        {       if(IsEntity) return (Entity*)Interface;
                else return NULL;
        }
        Relation* GetRelation()
        {       if(!IsEntity) return (Relation*)Interface;
                else return NULL;
        }
        ER_Diagram * GetMyERD()
        {       if(IsEntity) return GetEntity()->GetMyERD();
                else return GetRelation()->GetMyERD();
        }
        DataObject* GetDataObject()  {return GetMyERD()->GetInterface();}
        TERDDrawWindow* GetERDWin()  {return GetMyERD()->GetERDWin();}
        void SetERDWin(TERDDrawWindow* erdWin)  {GetMyERD()->SetERDWin(erdWin);}

        void SetLabel(char * label)  { GetDataObject()->SetName(label);}
        DataType GetType()  {return GetDataObject()->GetType();}
        char* GetLabel() {return (char*)GetDataObject()->GetName().data();}
        TPoint * GetCenter(Entity* entity)
        {       Point* p = entity->GetCenter();
                return new TPoint(p->X, p->Y);
        }
        TPoint * GetStart(){ return Start;}
        TPoint * GetEnd() { return End;}
        RWDlistCollectables* GetRelationList(Entity* entity) {return entity->GetRelationList();}
         // Draw ERD
        void ConstructEntity();
        void ConstructRelation();
        void DrawRelation(Entity* source, Entity* dest);
}; // end of VisualRepERD
#endif
```

E.2. The data repository

1) DFD tree manager

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  DFDManager
// Data repository interface that contain DFD tree and
// pointers to dictionaries and data graph
//        SuperClass: RWCollectable
//        File:     manager.h
//        Author: Jackson He
//        Date:    03/93
//        Language:C++
//        Modification notes:

#if !defined (_Manager)
#define _Manager
#include "drawogl.h"
#include "define.h"
#include "dhword.h"
_CLASSDEF (HyperTextWindow)

//***** DFDManager ******//
class DFDTreeManager: public RWCollectable
{ RWDECLARE_COLLECTABLE(DFDTreeManager)
  protected:
        RWCString ProjName;
        DFD* Root;
        DFD * CurrentDFD;
        ProjectDictionary * PDictionary;
        DataDictionary * DDictionary;
        DataRelGraph * DGraph;
        HyperTextWindow* HyperWin;
  public:
        DFDTreeManager();
        DFDTreeManager(const RWCString & name);
        ~DFDTreeManager();
        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCollectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
        RWCString & GetProjName() { return ProjName;};
        ProjectDictionary * GetProjDictionary() { return PDictionary;};
        DataDictionary * GetDataDictionary() {return DDictionary;};
        DataRelGraph * GetDataGraph() {return DGraph;};
        void Reset();  // move CurrentNode to Root;
        void MoveTo(DFD * adfd);   // move CurrentNode to specific node
```

```
        DFD * GetRoot();
        DFD * GetCurrentDFD();
        DFD* ExplodeSubDFD(const RWCString & name, Process* parent);
                // explore to next level DFD from CurrentDFD
        void InsertDFD(DFD * adfd); // insert a DFD as a child of Current DFD
        void RemoveDFD(DFD * adfd); // remove a DFD from CurrentDFD children list
        DFD * FindDFD(const RWCString & name, DFD* start);
                // retrieval a DFD in the tree for a given name from start point on
        DFD * FindDFD(DFD* adfd); // call previous FindDFD to check in adfd is in current tree;
        RWDlistCollectables BrowseTree(); // list all tree node in certain order
        void PersistenceSave(char * filename); // persistence save to a file

        //methods for HyperWord
        void SetHyperWin(HyperTextWindow* hyperwin) {HyperWin=hyperwin;}
        DFDObject* GetObjFromHWord(HyperWord* hword);
        HyperWord* GetSelectedHWord();
        HyperWord* GetDFDHWord(DFDObject* dobj);
        HyperWord* ConnectDFDHWord(HyperWord* hword, DFDObject* dobj);
                // connect a hyperword with a dfd object
        RWBoolean DisconnectDFDHWord(DFDObject* dobj);
                // disconnect all links with the dfd object
        RWBoolean DisconnectDFDHWord(HyperWord* hword, DFDObject* dobj);
                // disconnect the link of given hyperword */

        // Operations on Current DFD
        DFDObject * GetCurrentObject();
        void SetCurrentObject(DFDObject * obj);
        DFDNode * AddNode(const RWCString & label, DFDType type, Point * center);
                //add DFDNode to DFD and Proj. Dictionary
        RWBoolean DeleteNode(DFDNode *obj);
                // remove DFDNode from Data and Proj. dictionary
        RWBoolean DeleteFlow(Flow * flow); // remove Flow from Proj and Data dictionary
        Flow * AddFlow(const RWCString & label, DFDNode *source, DFDNode *dest);
                // Add a flow to DFD and Proj. dictionary
        Flow *GetFlow(DFDNode *source, DFDNode *dest);
                // Get flow for given source and destination
        DFDNode *GetNode(const RWCString &label, DFDType type);
        Flow * GetFlow(const RWCString &label);
        RWDlistCollectables * GetOutFlows(DFDNode *node); // Get outflow for a node
        RWDlistCollectables * GetInFlows(DFDNode *node); // Get inflow for a node
        RWDlistCollectables * GetNodeList(); // Get all nodes in this DFD
        RWDlistCollectables * GetFlowList();
        RWCString & GetDFDName();
        void SetDFDName(const RWCString & name);
        Process * GetParent();
        DFD* GetParentDFD();
        void AddDFDChild(Process *proc, DFD *child);
        void RemoveDFDChild(DFD *child);
        int NumChildren();
        RWDlistCollectables * GetChildren();
}; // end of DFDTreeManager
```

2) The dictionary class family

```
//Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  Dictionary
// objects for Project dictionary, data dictionary
// and their entries
//        SuperClass: DFDSet
//        File:     dictnary.h
//        Author: Jackson He
//        Date:    03/93
//        Language:C++
//        Modification notes:

#if !defined (_Dictionary)
#define _Dictionary

#include "define.h"
#include "dfdset.h"

// ******* ProjectDictionary *******//
class ProjectDictionary : public DFDSet
{  RWDECLARE_COLLECTABLE(ProjectDictionary)
 protected:
        DFDTreeManager* Interface;
 public:
        ProjectDictionary();
        ProjectDictionary(DFDTreeManager* interface);
         ~ProjectDictionary();
        DFDTreeManager* GetInterface() { return Interface;}
        ProjEntry * InsertDFDObject(DFDObject * dfdobj);
        ProjEntry * InsertProjEntry(ProjEntry * pentry);
        void RemoveDFDObject(DFDObject * dfdobj);
        void RemoveProjEntry(ProjEntry * pentry);
        ProjEntry * FindProjEntry(DFDObject * dfdobj);

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of ProjectDictionary

// ****** ProjEntry  ******//
class ProjEntry : public RWCollectable
{
 RWDECLARE_COLLECTABLE(ProjEntry)
 protected:
        RWCString  Label;
        DFDType  Type; // Type can be: EXT, PRO, STR, DF
        DescriptionCard * Description;
        RWDlistCollectables *  Links;
        Boolean CanMerge(DFDObject* dfdobj);
```

```
public:
        ProjEntry();
        ProjEntry(const RWCString & label, DFDType type);
        ~ProjEntry();

        RWCString& GetLabel() {return Label;};
        Boolean SetLabel(const RWCString & label, DFDObject* dfdobj);
        DFDType  GetType();
        DFDObject* InsertLinks(DFDObject * dfdobj);
        DFDObject* RemoveLinks(DFDObject * dfdobj);
        RWBoolean IsLinkEmpty();
        RWDlistCollectables * GetLinks();
        RWCString& GetDescript();
        void  SetDescript( const RWCString &  descript);
        void CleanUp(); // clean links before delete the entry

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCollectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;

}; // end of ProjEntry

//**** DescriptionCard *******//
class DescriptionCard : public RWCollectable
{ RWDECLARE_COLLECTABLE(DescriptionCard)
   protected:
        RWCString  Descript;
   public:
        DescriptionCard();
        DescriptionCard(const RWCString & descript);
         ~DescriptionCard();
         RWCString & GetDescript();
         void   SetDescript( const RWCString &  descript);

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of DescriptionCard

//******* DataDictionary ********//
class DataDictionary : public DFDSet
{  RWDECLARE_COLLECTABLE(DataDictionary)
   protected:
```

```
        DFDTreeManager * Interface;
  public:
        DataDictionary();
        DataDictionary(DFDTreeManager * interface);
        ~DataDictionary();
        DFDTreeManager * GetInterfacce() { return Interface;};
        DataEntry * InsertDataObject(DataObject * dataobj);
        DataEntry * InsertDataEntry(DataEntry * dentry);
        void RemoveDataObject(DataObject * dataobj);
        DataEntry * FindDataEntry(DataObject * dataobj);
        DataEntry * FindDataEntry(const RWCString& name, int type);
        RWDlistCollectables * listDictionary(); // list  all entires in dictionary

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) ;
}; // end of DataDictionary

//****** DataEntry *******//
class DataEntry : public RWCollectable
{RWDECLARE_COLLECTABLE(DataEntry)
 protected:
        RWCString Name;
        DataType Type; // Type can be: DataFlow, DataStore, Record, Element
        int isKey;
        int isCombKey;
        int CombKey;
        DataObject * DObject;
        RWCString Attrib;
        int Length;
        DescriptionCard* Script;
 public:
        DataEntry();
        DataEntry(const RWCString & name, DataType type,
                const RWCString& attr, int l, const RWCString& s, DataObject * dobj);
        ~DataEntry();
        DataObject * GetDataObject() {return DObject;};
        void SetName(const RWCString &);
        RWCString & GetName();
        DataType GetType();
        DataDictionary* GetDataDictionary();
        void CleanUp(); // clean up before deleting
        RWCString& GetAttrib() { return Attrib;}
        void SetAttrib(const RWCString& attr) { Attrib = attr;}
        int GetLength(){return Length;}
        void SetLength(int l){Length = l;}
        RWCString& GetScript() { return Script->GetDescript();}
        void SetScript(const RWCString& s) { Script->SetDescript(s);}
```

```
        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCol!ectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of DataEntry
#endif
```

3) The Data relation graph and data object

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)
// Object for data relation graph and basic data objects
//      SuperClass: RWCollectable
//      File:   data_rel.h
//      Author: Jackson He
//      Date:   03/93
//      Language:C++
//      Modification notes:

#if !defined (_DataRel)
#define _DataRel

#include "define.h"
#include "dictnary.h"
_CLASSDEF (DataWindow)
//DataRelGraph
class DataRelGraph : public RWCollectable
{ RWDECLARE_COLLECTABLE(DataRelGraph)
  protected:
        RWDlistCollectables * Nodes; // list for all the DataNodes in graph - roots of data trees
        DFDTreeManager * Interface; // pointer back to DFDTreeManager
        DataObject* HyperParent;
        // HyperParent serve as parent for DataItems that created without parent
        //this DataObject will not be in DataDictionary
public:
        DataRelGraph();
        DataRelGraph(DFDTreeManager * interface);
        ~DataRelGraph();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;

        DataNode *AddDataNode(const RWCString &name, DataType type,
```

```
                                ChildItem *child, ConceptObject * interface);
        // create and add a DataNode to data dictioanry
        DataItem *AddDataItem(const RWCString &name, DataType type,int i,
          RWBoolean iskey, RWBoolean iscombkey, int combkey,
          DataObject *parent, ChildItem *child,
          const RWCString& attr, int l, const RWCString& s);
                // Create and add a DataItem to data dictionary as i-th child of its parent
        void DeleteDataObject(DataObject *obj); // delete a dataobject from graph and data dicctionary
        void DeleteDataNode(DataNode* node, ConceptObject* cobj);
        RWDlistCollectables *GetChildren(DataObject *node); // Get children list of a node
        RWDlistCollectables *GetParents(DataObject *node); // get parents list of a node
        void ConnectObjects(DataObject * source, ChildItem* child, int i);
                // connect two data object with key attributesdest is i-th child of source
        ChildItem* DisconnectObjects(DataObject *source, ChildItem *dest);
        ChildItem* DisconnectObjects(DataObject *source, DataObject* dest);
                // disconncet two dataobject
        DataNode* FindDataNode(DataNode* obj)
                { return (DataNode*)Nodes->find((RWCollectable*)obj);}
        DataObject* FindDataObject(DataObject* obj);
        DataObject* FindDataObject(const RWCString& name, int type);
        DataDictionary * GetDataDictionary();
        DFDTreeManager* GetInterface() { return Interface;};
        DataObject* GetHyperParent() {return HyperParent;};
}; // end of DataRelGraph


// ********* ChildItem ******************//
class ChildItem: public RWCollectable
{RWDECLARE_COLLECTABLE(ChildItem)
 protected:
        DataObject * Child;
        int isKey;
        int isCombKey;
        int CombKey;
        Entity* MyEntity;
 public:
        ChildItem();
        ChildItem(DataObject* obj, RWBoolean iskey,
                RWBoolean iscombkey, int combkey);
        ~ChildItem();

        RWBoolean IsKey(){return isKey;};
        void SetKey(RWBoolean key) {isKey = key;};
        RWBoolean IsCombKey(){ return isCombKey;};

        void SetCombKey(RWBoolean iscombkey, int combkey)
        {       isCombKey = iscombkey;
                if(isCombKey) CombKey = combkey;
                else CombKey = 0;
        }

        int GetCombKey(){ return CombKey;};
```

```
        DataObject * GetChild(){ return Child;};
        Entity* GetMyEntity(){return MyEntity;}

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCollectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of ChildItem


// ******* DataObject  *******//
class DataObject : public RWCollectable
{ RWDECLARE_COLLECTABLE(DataObject)
protected:
        DataEntry * Entry;
        RWDlistCollectables *  Children;
        RWDlistCollectables *  Parents;
        DataRelGraph * Container;
        ER_Diagram* MyERD;
public:
        DataWindow * DataWin;
        DataObject();
        DataObject(const RWCString & name, DataType type,
          const RWCString& attr, int l, const RWCString& s,
          DataRelGraph * container);
        ~DataObject();

        //Basic opertations
        RWCString & GetName();
        void  SetName(const RWCString & name); // set a new name
        DataObject *SetLabel(const RWCString & label); // change name, make it unique.
        void DumpChildren();
        void DumpParents(); // cut off children and parents connections
        DataType GetType();
        DataEntry * GetDataEntry();
        void SetDataEntry(DataEntry * entry);
        RWDlistCollectables *  GetChildren();
        void MergeChildList(DataObject* obj);
        RWDlistCollectables * GetParents();
        void MergeParentList(DataObject* obj);
        ER_Diagram* GetMyERD(){return MyERD;}

        // List operations
        void InsertAChild(ChildItem* child); // insert a child with key attributes
        void InsertAChildAfter(int index, ChildItem* child);
        void InsertAParent(DataObject * dataobject);
        ChildItem* RemoveAChild(ChildItem * dataobject, Boolean fromERD);
```

```
        void RemoveAParent(DataObject * dataobject);
        ChildItem * FindAChild(DataObject * adata);
        DataObject * FindAParent(DataObject * adata);
        DataObject * GetFirstParent() { return (DataObject*) Parents->first();}
        RWDlistCollectables  Connectionlist(); // find all the connected Nodes with this node
          // directly and indirectly;
        DataRelGraph * GetContainer();
        int GetNumParents() { return Parents->entries();};
        RWDlistCollectables SubList();
        ChildItem * GetChildItem(DataObject* parent);

        // key operations
        RWBoolean IsKey(DataObject * parent);
        void SetKey(DataObject* parent, RWBoolean iskey);
        RWBoolean IsCombKey(DataObject * parent);
        void SetCombKey(DataObject* parent, RWBoolean iscombkey, int combkey);
        int GetCombKey(DataObject* parent);
        void CleanUp();

        // attributes
         RWCString& GetAttrib() { return Entry->GetAttrib();}
        void SetAttrib(const RWCString& attr) { Entry->SetAttrib(attr);}
        int GetLength(){return Entry->GetLength();}
        void SetLength(int l){Entry->SetLength(l);}
        RWCString& GetScript() { return Entry->GetScript();}
        void SetScript(const RWCString& s) { Entry->SetScript(s);}

        // Inherited from class "RWCollectable Persistance methods
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCollectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
};  //end of DataObject

// ****** DataNode *********//
class DataNode : public DataObject
{ RWDECLARE_COLLECTABLE(DataNode)
   protected:
        ConceptObject * Interface;
  public:
        DataNode();
        DataNode(const RWCString & name, DataType type,
                DataRelGraph * container, ConceptObject* interface);
        ~DataNode();

        ConceptObject * GetInterface() { return Interface;};
        DFDObject * GetDFDObject();
        void SetLabel(const RWCString & label, ConceptObject* cobj);
```

```
        void SetConceptData(ConceptObject* cobj, DataNode* dobj);
        void InsertAParent(ConceptObject * cobject) { Parents->insert((RWCollectable*) cobject);};
        void MergeParentList(DataNode*);
        void RemoveAParent(ConceptObject * cobject)
                {Parents->removeReference((RWCollectable*)cobject);};
        void CleanUp();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; //end of DataNode

//******** DataItem *************//
class DataItem : public DataObject
{ RWDECLARE_COLLECTABLE(DataItem)
public:
        DataItem();
        DataItem(const RWCString & name, DataType type,
                const RWCString& attr, int l, const RWCString& s,
                DataRelGraph * container);
        ~DataItem();
}; // end of DataItem

// ********* DataElement ********//
class DataElement : public DataItem
{ RWDECLARE_COLLECTABLE(DataElement)
 public:
        DataElement();
        DataElement(const RWCString & name,
                const RWCString& attr, int l, const RWCString& s,
        DataRelGraph * container);
        ~DataElement();
}; //end of DataElement

// ******** DataRecord ********//
class DataRecord: public DataItem
{ RWDECLARE_COLLECTABLE(DataRecord)
 public:
        DataRecord();
        DataRecord(const RWCString & name,
        const RWCString& attr, int l, const RWCString& s,
                DataRelGraph * container);
        ~DataRecord();
};  // end of DataRecord
#endif
```

4) The DFD class family

```
// Copyright (C) 1993 by University of Hawaii
```

```
// Hyper Analysis Toolkit (R)  DFD
// Objects of all DFD and DFD elements
//        SuperClass: RWCollectable
//        File:     dfd.h
//        Author: Jackson He
//        Date:     03/93
//        Language:C++
//        Modification notes:

#if !defined (_DFD)
#define _DFD

#include "define.h"
#include "data_rel.h"
#include "dictnary.h"
#include "manager.h"
#include "visual.h"
#include "siminfor.h"

 // ******  DFD ******//
class DFD : public RWCollectable
{ RWDECLARE_COLLECTABLE(DFD)
   protected:
        RWDlistCollectables * Nodes;  // list of DFDNodes
        RWDlistCollectables * Flows;  // list for Flow
        RWDlistCollectables * Children; // list of children DFDs
        RWBoolean HasParent;
        Process * Parent; // list containes a pointer to Process from which it is explored
        DFDTreeManager * Manager; // pointer to DFDTreeManager
        int Counter; // counter for visual, need to be saved
        int Level; // Level in DFD Tree
        ConceptDFD * CDFD; // Pointer to buddy concept DFD
        VisualDFD * VDFD; // pointer to buddy visual DFD
        DFDObject * CurrentObject; // pointer to current DFDobject

        Boolean AdjustInterfaceNode(InterfaceNode* inf); // called by ReAllienInterface()
        SimRun* simRun;
   public:
        DFD();
        DFD(const RWCString & name, int level,Process* parent, DFDTreeManager * manager);
        ~DFD();
        int GetLevel(){return Level;}
        int GenerateId();
        SimRun* GetsimRun(){return simRun;}
        Id AddId(); // return a combined Id for display
        void SetCounter(int c) { Counter = c;};
        DFDObject * GetCurrentObject() { return CurrentObject;}
        void SetCurrentObject(DFDObject * obj) {CurrentObject = obj;}
        DFDNode * AddNode(const RWCString & label, DFDType type, Point * center);
                //add DFDNode to DFD and Proj. Dictionary
        DFDNode *DFD::AddNode(DFDNode *node); // Adding operation
        RWBoolean DeleteNode(DFDNode *obj);  // remove DFDNode from Data and Proj. dictionary
```

```
        RWBoolean DeleteFlow(Flow * flow); // remove Flow from Proj and Data dictiontioary
        Flow * AddFlow(const RWCString & label, DFDNode *source, DFDNode *dest);
                // Add a flow to DFD and Proj. dictionary
        Flow *GetFlow(DFDNode *source, DFDNode *dest); // Get flow for given source and
destination
        DFDNode *GetNode(const RWCString &label, DFDType type);
        Flow * GetFlow(const RWCString &label);
        RWDlistCollectables * GetOutFlows(DFDNode *node); // Get outflow for a node
        RWDlistCollectables * GetInFlows(DFDNode *node); // Get inflow for a node
        RWDlistCollectables * GetNodeList(); // Get all nodes in this DFD
        RWDlistCollectables * GetFlowList() { return Flows;};
        RWCString & GetName();
        void SetName(const RWCString & name);
        Process * GetParent();
        DFD* GetParentDFD();
        void AddDFDChild(Process *proc, DFD *child);
        void RemoveDFDChild(DFD *child);
        int NumChildren() { return Children->entries();};
        DFDTreeManager *GetManager();
        void SetManager(DFDTreeManager * manager);
        DataRelGraph * GetDataGraph() { return Manager->GetDataGraph();};
        ProjectDictionary * GetProjDictionary();
        DataDictionary * GetDataDictionary();
        RWDlistCollectables * GetChildren() { return Children;};
        void ReAllienInterface();
                // check for connection to its parent and set interface node accordingly
        Boolean HasInterfaceNode(DFDNode* node, Boolean isFrom);
                // test if a node has its corresponding interface node test method
        void prntDFDList();  // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCollectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of DFD

//********** DFDObject **************//
 class DFDObject : public RWCollectable
{ RWDECLARE_COLLECTABLE(DFDObject)
   protected:
        ConceptObject * CObject;
        VisualObject * VObject;
        DFD * Container;
   public:
        DFDObject();
        DFDObject(DFD * container); // create conceptual and visual objects
        ~DFDObject();

        Id  GetDisId(); //void SetDisId(Id id);
```

```
        RWCString& GetLabel();
        void SetLabel(const RWCString & label);
        DFDType GetType();
        DFD * GetContainer();
        VisualObject * GetVisual()          {return VObject;};
        ConceptObject * GetConcept() {return CObject;};
        ProjEntry* GetProjEntry();
        RWCString &  GetDescript();
        void  SetDescript( const RWCString &  descript);
        void CleanUpVisual();  // clean up visual part before deleting the whole project

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCollectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of DFDObject

//******** DFDNode **********//
 class DFDNode : public DFDObject
{  RWDECLARE_COLLECTABLE(DFDNode)
   protected:
        RWDlistCollectables * InFlow;
        RWDlistCollectables * OutFlow;
        RWDlistCollectablesIterator * InList;
        RWDlistCollectablesIterator * OutList;
   public:
        DFDNode();
        DFDNode(DFD * container);
        ~DFDNode()
        RWDlistCollectables * GetInFlow();
        RWDlistCollectables * GetOutFlow();
        VisualNode* GetVisualNode() {return (VisualNode*) GetVisual();};
        ConceptNode * GetConceptNode() {return (ConceptNode*) GetConcept();};
        Point * GetCenter() { return GetVisualNode()->GetCenter();};
        void SetCenter(Point* Pt) { GetVisualNode()->SetCenter(Pt);};
        Flow * GetNextInFlow();
        Flow * GetNextOutFlow();
        Flow * InsertAnInFlow(Flow * aflow);
        Flow * InsertAnOutFlow(Flow * aflow);
        void  RemoveAnInFlow(Flow * aflow);
        void  RemoveAnOutFlow(Flow * aflow);
        Flow * FindAnInFlow(Flow * aflow);
        Flow * FindAnOutFlow(Flow * aflow);
        DbList* BuildAssignList(); // retrieve outflows and build assignment list for simulation
        virtual void SetSimDescript(char* str);
        virtual RWCString& GetSimDescript();
        virtual RWCString* ListSimStats();
```

```
        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
};  // End of DFDNode


//********** Process ****************//
class Process: public DFDNode
{ RWDECLARE_COLLECTABLE(Process)
protected:
        DFD* NextLevel;
        Boolean HasNextLevel;
        SimQ* simQ;
        SimAct* simAct;
public:
        Process();
        Process(Id id, const RWCString & label, Point * center, DFD * container);
        ~Process();

        SimQ* GetsimQ(){return simQ;}
        SimAct* GetsimAct(){return simAct;}
        DFD * GetNextLevel();
        void SetNextLevel(DFD * dfd);
        void RemoveNextLevel();
        void ResetNextLevel() { SetNextLevel(NULL);}
        virtual void SetSimDescript(char* str);
        virtual RWCString& GetSimDescript();
        virtual RWCString* ListSimStats();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // End of Process

//******* Store ********//
class Store : public DFDNode
{ RWDECLARE_COLLECTABLE(Store)
protected:
        SimResource* simResource;
public:
        Store();
        Store(Id id, const RWCString & label, Point* center, DFD * container);
        ~Store();

        DataNode * GetData();
```

```
        void SetData(DataNode * data);
        SimResource* GetsimResource(){return simResource;}
        virtual void SetSimDescript(char* str);
        virtual RWCString& GetSimDescript();
        virtual RWCString* ListSimStats();

        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // End of Store


// ******* Flow ********//
class Flow : public DFDObject
{ RWDECLARE_COLLECTABLE(Flow)
  protected:
        DFDNode * Source;
        DFDNode * Destination;
        SimFlow* simFlow;
  public:
        Flow();
        Flow(Id id, const RWCString & label, RWDlistCollectables* path,
                DFDNode * s, DFDNode * d, DFD* container);
        ~Flow();

        SimFlow * GetsimFlow(){return simFlow;}
        DataNode * GetData();
        void SetData(DataNode * data);
        VisualFlow * GetVisualFlow();
        void SetVisualFlow(VisualFlow * visualflow);
        DFDNode * GetSource() {return Source;};
        DFDNode* GetDestination() { return Destination;};
        ConceptFlow * GetConceptFlow() {return (ConceptFlow*) GetConcept();};

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of Flow

// ****** ExtEntity *******//
class ExtEntity: public DFDNode
{ RWDECLARE_COLLECTABLE(ExtEntity)
protected:
        SimSrc* simSrc;
        SimSink* simSink;
  public:
        ExtEntity();
        ExtEntity(Id id, const RWCString & label, Point* center, DFD * container);
```

```
        ~ExtEntity();

        SimSrc* GetsimSrc(){return simSrc;}
        SimSink* GetsimSink(){return simSink;}
        virtual void SetSimDescript(char* str);
        virtual RWCString& GetSimDescript();
        virtual RWCString* ListSimStats();

        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of ExtEntity

//****** InterfaceNode ********//
class InterfaceNode : public DFDNode
{ RWDECLARE_COLLECTABLE(InterfaceNode)
protected:
        SimSrc* simSrc;
        SimSink* simSink;
  public:
        InterfaceNode();
        InterfaceNode(Id id, const RWCString & label, Point* center, DFD * container,
                        DFDNode* intnode);
        ~InterfaceNode(){delete simSrc; delete simSink;};
        ConceptINFNode* GetCINFNode() { return (ConceptINFNode*) CObject;}
        DFDNode * GetInterfaceNode();
        void SetInterfaceNode(DFDNode* intnode);

        // Simulation-related operations
        SimSrc* GetsimSrc(){return simSrc;}
        SimSink* GetsimSink(){return simSink;}
        virtual void SetSimDescript(char* str);
        virtual RWCString& GetSimDescript();
        virtual RWCString* ListSimStats();

        // Persistent methods
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of InterfaceNode
#endif
```

5) The Conceptual DFD class family

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  Concept
// contains all the conceptual objects of DFD
//      SuperClass: RWCollectable
//      File:   concept.h
//      Author: Jackson He
//      Date:   03/93
```

```
//      Language:C++
//      Modification notes:

#if !defined (_CONCEPT)
#define _CONCEPT
#include "define.h"
#include "data_rel.h"
#include "manager.h"
#include "dfd.h"

 //*****  ConceptDFD *******//
class ConceptDFD : public RWCollectable
{ RWDECLARE_COLLECTABLE(ConceptDFD)
protected:
        DFD *Interface; // pointer back to DFD
        RWCString Name; // DFDName
  public:
        ConceptDFD();
        ConceptDFD( const RWCString & name, DFD *interface);
        ~ConceptDFD();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCollectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;

        // basic operations
        RWCString & GetName();
        void SetName(const RWCString & name);
        ConceptFlow *GetFlow(ConceptNode *source, ConceptNode *dest);
                //return conceptfloe for given source and dest.
        RWDlistCollectables GetOutFlows(ConceptNode *node);  //list outflows of a node
        RWDlistCollectables GetInFlows(ConceptNode *node);   // list Inflows of a node
        RWDlistCollectables GetAllFlows(ConceptNode *node); // list both in and out flows of a node
}; // end of ConceptDFD

 // ******** ConceptObject ********//
class ConceptObject : public RWCollectable
{ RWDECLARE_COLLECTABLE(ConceptObject)
protected:
        ProjEntry * Entry; // entry to proj. dictionary.
        DFDObject * Interface; // pointer back to it buddy DFDObject
  public:
        ConceptObject();
        ConceptObject(const RWCString & label, DFDType type,DFDObject * interface);
        ~ConceptObject();
        // Inherited from class "RWCollectable":
```

```
        unsigned        binaryStoreSize() const;
        int             compareTo(const RWCollectable*) const;
        RWBoolean       isEqual(const RWCollectable*) const;
        unsigned        hash() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
        //ConceptObject methods
        RWCString& GetLabel();
        void SetLabel(const RWCString & label);
        DataType GetType();
        RWCString& GetDisId(){return Interface->GetDisId();}
        DFDObject * GetInterface();
        void SetEntry(ProjEntry * entry);
        void ResetEntry(ProjEntry* entry) {Entry = entry;}
        ProjEntry * GetProjEntry();
        ProjectDictionary * GetProjDictionary();     // return proj. dictionary
        DataDictionary * GetDataDictionary();   // return data dictionary
        DataRelGraph * GetDataGraph()   // retain data relation graph
        { return Interface->GetContainer()->GetManager()->GetDataGraph();}
        RWCString& GetDescript();
        void   SetDescript( const RWCString &  descript);
}; //end of ConceptObject

// ********** ConceptNode **********//
class ConceptNode : public ConceptObject
{ RWDECLARE_COLLECTABLE(ConceptNode)
  public:
        ConceptNode();
        ConceptNode(const RWCString & label, DFDType type, DFDObject * interface);
        ~ConceptNode();
        RWDlistCollectables * GetInFlow();
        RWDlistCollectables * GetOutFlow();
        Flow * GetNextInFlow();
        Flow * GetNextOutFlow();
        void InsertAnInFlow(Flow * aflow);
        void InsertAnOutFlow(Flow * aflow);
        void RemoveAnInFlow(Flow * aflow);
        void RemoveAnOutFlow(Flow * aflow);
        Flow * FindAnInFlow(Flow * aflow);
        Flow * FindAnOutFlow(Flow * aflow);
};   // end of ConceptNode

//****** ConceptProcess ********//
class ConceptProcess: public ConceptNode
{ RWDECLARE_COLLECTABLE(ConcpetProcess)
  public:
        ConceptProcess();
        ConceptProcess(const RWCString & label, DFDObject * interface);
        ~ConceptProcess();
}; // end of ConceptProcess
```

```
//******** ConceptStore *********//
class ConceptStore : public ConceptNode
{ RWDECLARE_COLLECTABLE(ConceptStore)
  protected:
        DataNode * Data;
  public:
        ConceptStore();
        ConceptStore(const RWCString & label, DFDObject * interface);
        ~ConceptStore();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;

        DataNode * GetData();
        void SetData(DataNode * data);
}; // end of ConceptStore

// ************ ConceptFlow ***********//
class ConceptFlow : public ConceptObject
{ RWDECLARE_COLLECTABLE(ConceptFlow)
  protected:
        DataNode * Data;
public:
        ConceptFlow();
        ConceptFlow(const RWCString & label, DFDObject * interface);
        ~ConceptFlow();
        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
        DataNode * GetData();
        void SetData(DataNode * data);
}; // end of ConceptFlow

// ********* ConceptExtEntity ***********//
class ConceptExtEntity: public ConceptNode
{  RWDECLARE_COLLECTABLE(ConceptExtEntity)
  public:
        ConceptExtEntity();
        ConceptExtEntity(const RWCString & label, DFDObject * interface);
        ~ConceptExtEntity();
}; // end of ConceptExtEntity

// ******** ConceptINFNode *********//
class ConceptINFNode: public ConceptNode
```

```
{ RWDECLARE_COLLECTABLE(ConceptINFNode)
  protected:
        DFDNode * IntNode;
        Boolean HasInterface;
  public:
        ConceptINFNode();
        ConceptINFNode(const RWCString & label, DFDObject * interface, DFDNode* intnode);
        ~ConceptINFNode();
        DFDNode * GetInterfaceNode();
        void SetInterfaceNode(DFDNode* intnode);
         // persistent methods for RWCollectable
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of ConceptINFNode
#endif
```

6) The Visual DFD class family

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  Visual
// Contains all the objects for DFD visual features
//        SuperClass: RWCollectable
//        File:     visual.h
//        Author: Jackson He
//        Date:    03/93
//        Language:C++
//        Modification notes:

#if !defined (_Visual)
#define _Visual

#include "define.h"
_CLASSDEF (VisualRepDFD)

 // ***** VisualDFD ****** //
class VisualDFD : public RWCollectable
{ RWDECLARE_COLLECTABLE(VisualDFD)
   protected:
        DFD *Interface;
   public:
        VisualDFD();
        VisualDFD(DFD *interface);
        ~VisualDFD();
        void SetInterface(DFD *inter) { Interface = inter; }
        DFD *GetInterface() { return Interface; }
         VisualFlow * GetFlow(VisualNode *source, VisualNode *dest);
        RWDlistCollectables * GetOutFlows(VisualNode * node);
        RWDlistCollectables * GetInFlows(VisualNode * node);
        RWDlistCollectables GetAllFows(VisualNode * node);
```

```
        // Inherited from class "RWCollectable" for persistent store and retrieval
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of VisualDFD

//***** VisualObject *********//
 class VisualObject : public  RWCollectable
{  RWDECLARE_COLLECTABLE(VisualObject)
protected:
        Id ID;
        DFDObject * Interface;
        VisualRepDFD * VRep; graph object from VisualRepDFD or other graph tool
public:
        VisualObject();
        VisualObject(Id id, DFDObject * interface);
        ~VisualObject();
        VisualRepDFD * GetVRep();
        void SetVRep(VisualRepDFD* vrep) {VRep=vrep;}
        Id GetId();
        void SetId(Id id);
        DFDObject * GetInterface() {return Interface;};
        DFDType GetType();
        RWCString * GetLabel();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of VisualObject

// **** VisualNode ****** //
class VisualNode: public  VisualObject
{ RWDECLARE_COLLECTABLE(VisualNode)
   protected:
        Point * Center;
   public:
        VisualNode();
        VisualNode(Id id, Point* center, DFDObject * interface);
        ~VisualNode();
        void SetCenter(Point * center);
        Point * GetCenter();
        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
```

```
        void            saveGuts(RWvostream&) const;
}; // end of VisualNode

//***** VisualProcess******//
class VisualProcess: public  VisualNode
{ RWDECLARE_COLLECTABLE(VisualProcess)
  public:
        VisualProcess(){};
        VisualProcess(Id id, Point* center, DFDObject * interface);
        // Get label from interface with GetLabel()
        ~VisualProcess(){};
};  // end of VisualProcess

//**** VisualStore *****//
class VisualStore: public  VisualNode
{RWDECLARE_COLLECTABLE(VisualStore)
  public:
        VisualStore(){};
        VisualStore(Id id, Point * center, DFDObject * interface);
        ~VisualStore(){};
}; // end of VisualStore

// ***** VisualExtEntity ******/
class VisualExtEntity: public  VisualNode
{ RWDECLARE_COLLECTABLE(VisualExtEntity)
  public:
        VisualExtEntity(){};
        VisualExtEntity(Id id, Point * center, DFDObject * interface);
        ~VisualExtEntity(){};
}; // end of VisualExtEntity

//****** VisualINFNode *******
class VisualINFNode: public  VisualNode
{  RWDECLARE_COLLECTABLE(VisualINFNode)
  public:
        VisualINFNode(){};
        VisualINFNode(Id id, Point * center, DFDObject * interface);
        ~VisualINFNode(){};
}; // end of VisualINFNode

// ***** VisualFlow *******
class VisualFlow : public VisualObject
{ RWDECLARE_COLLECTABLE(VisualFlow)
  protected:
        RWDlistCollectables * Path;
        void FillPtArray();
        void FillPath() const;
  public:
        Point* PtArray[5];
        VisualRepDFD * ConnectorHandle[4];
        VisualRepDFD* FLabel; // pointer to floating label
        VisualFlow();
```

```
        VisualFlow(Id id, RWDlistCollectables * path, DFDObject* interface);
        ~VisualFlow();
        void InsertAPoint(Point * apoint);
        void RemoveAPoint(Point * apoint);
        Point * GetStart();
        Point * GetEnd();
        void SetStart(Point * point);
        void SetEnd(Point * point);
        void SetNth(int ind, Point * point);
        Point * GetNth(int ind);
        void InsertB4Nth(int ind, Point * point);
        void DeleteB4Nth(int ind);
        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWFile&);
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWFile&) const;
        void            saveGuts(RWvostream&) const;
}; // end of VisualFlow
#endif
```

7) The ERD class family

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  ERD
// Objects of all ERD and ERD elements
//      SuperClass: RWCollectable
//      File:    erd.h
//      Author: Jackson He
//      Date:    03/93
//      Language:C++
//      Modification notes:

#ifndef _ERD.H
#define _ERD.H

#include "define.h"
#include "data_rel.h"
#include "visual.h"

//******** ER_Diagram ***********//
class ER_Diagram: public RWCollectable
{ RWDECLARE_COLLECTABLE(ER_Diagram)
  protected:
        DataObject * Interface;
        DbList* EntityList;
        DbList* RelationList;  // two lists hold entities and relations
        TERDDrawWindow * ERDWin; // pointer to drawing window
        RWCString Name;
  public:
        int Count; // counter for number nodes in the graph
        RWCollectable* CurObject; // pointer to current object
```

```
        ER_Diagram();
        ER_Diagram(DataObject* interface);
        ~ER_Diagram();
        //basic opertations
        DataObject* GetInterface(){return Interface;}
        DbList* GetEntityList(){return EntityList;}
        DbList* GetRelationList(){return RelationList;}
        TERDDrawWindow* GetERDWin(){return ERDWin;}
        void SetERDWin(TERDDrawWindow* drwin){ERDWin = drwin;}
        RWCString & GetERDName()
        {       if(Interface!=NULL)
                return Interface->GetName();
                else
                {       Name="Top Level ERD";
                        return Name;
                }
        }
        Entity* AddEntity(const RWCString &name, DataType type,
                RWBoolean iskey, RWBoolean iscombkey, int combkey, Point* center);
        Entity* AddEntity(ChildItem* child);  // Add new data to dictionary
        Relation* AddRelation(Entity* source, Entity* dest, const RWCString& label,
                const RWCString& s_rel, const RWCString& d_rel,
                const RWCString& s);
        Entity* DeleteEntity(Entity* entity, Boolean fromERD);
        Entity* DeleteEntity(ChildItem* child, Boolean fromERD);
        void DeleteRelation(Relation* rel);
         //Persistence methods from RWCollectable
        unsigned        binaryStoreSize() const;
        void restoreGuts(RWFile&);
        void restoreGuts(RWvistream&);
        void saveGuts(RWFile&) const;
        void saveGuts(RWvostream&) const;
}; // end of ER_Diagram

// ******** Entity **************//
class Entity : public RWCollectable
{  RWDECLARE_COLLECTABLE(Entity)
   protected:
        DataObject* Interface;
        DbList* RelationList;
        Point * Center;
        VisualRepERD* VEntity;
   public:
        Entity();
        Entity(DataObject* interface);
        ~Entity();
        //Basic operations
        DataObject * GetInterface(){return Interface;}
        DbList* GetRelationList(){return RelationList;}
        Point * GetCenter(){return Center;}
        void SetCenter(Point * center){ Center = new Point(center->X, center->Y);}
```

```
        VisualRepERD* GetVEntity(){return VEntity;}
        void SetVEntity(VisualRepERD* ventity) {VEntity = ventity;}
        char* GetName(){return (char*){ Interface->GetName().data();}
        ER_Diagram* GetMyERD(){return Interface->GetMyERD();}
        RWCString& GetScript() {return Interface->GetScript();}

        // ADD & Delete relation
        Relation * AddRelation(Relation* rel);
        Relation * RemoveRelation(Relation* rel);
        Relation * FindARelation(Relation* rel);
        Relation * FindARelation(Entity* entity);

        //Persistence methods from RWCollectable
        unsigned        binaryStoreSize() const;
        int compareTo(const RWCollectable*) const;
        RWBoolean isEqual(const RWCollectable*) const;
        unsigned hash() const;
        void restoreGuts(RWFile&);
        void restoreGuts(RWvistream&);
         void saveGuts(RWFile&) const;
        void saveGuts(RWvostream&) const;
}; // end of Entity


// ******** Relation **********//
class Relation : public RWCollectable
{ RWDECLARE_COLLECTABLE(Relation)
   protected:
        RWCString Label;
        VisualRepERD* VRelation;
        RWCString Rel1;
        RWCString Rel2;
        DescriptionCard* Script;
        Entity * E1;
        Entity * E2;
public:
        Relation();
        Relation(Entity* e1, Entity* e2, const RWCString& label,
                const RWCString& rel1, const RWCString& rel2, const RWCString& s);
         ~Relation();
        //Basic Operations
        Entity * GetE1(){return E1;}
         Entity * GetE2(){return E2;}
        void SetE1(Entity* entity){E1 = entity;}
        void SetE2(Entity* entity){E2 = entity;}
        RWCString& GetLabel(){return Label;}
        void SetLabel(const RWCString& label) {Label=label;}
        VisualRepERD* GetVRelation(){return VRelation;}
        void SetVRelation(VisualRepERD* vrel) {VRelation = vrel;}
        RWCString& GetRel1(){return Rel1;}
        void SetRel1(const RWCString & rel1){Rel1 = rel1;}
        RWCString& GetRel2(){return Rel2;}
```

```
        void SetRel2(const RWCString & rel2){Rel2 = rel2;}
        ER_Diagram* GetMyERD(){return GetE1()->GetMyERD();}
        RWCString& GetScript(){return Script->GetDescript();}
        void SetScript(const RWCString& s){ Script->SetDescript(s);}

        //Persistence methods from RWCollectable
        unsigned        binaryStoreSize() const;
        int compareTo(const RWCollectable*) const;
        RWBoolean isEqual(const RWCollectable*) const;
        void restoreGuts(RWFile&);
        void restoreGuts(RWvistream&);
        void saveGuts(RWFile&) const;
        void saveGuts(RWvostream&) const;
}; // end of Relation
#endif
```

8) Simulation information classes

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  SimInfor
// Objects to hold simulation information
//      SuperClass: RWCollectable
//      File:    siminfor.h
//      Author: Jackson He 02/94
//      Language C++

#ifndef _SimInfor_H
#define _SimInfor_H

#include "define.h"
#define NumBranch       10
#define Weighted        1
#define Unweighted      0
#define Mini            0
#define Maxi            1
#define Mean            2
#define Std             3
#define  Obs            4
#define Sum             5

// ******** SimResult ********//
class SimResult : public RWCollectable
{ RWDECLARE_COLLECTABLE(SimResult)
 public:
        float Stats[2][6];
        RWCString Description;
         SimResult();
        ~SimResult(){};
        void ParseWtResult(char* str);  // parse time-weighted result
        void ParseUwResult(char* str); // parse unweighted result
        Boolean HasResult();
        RWCString& GenDescription();
```

```
        RWCString& AppendDescription(const RWCString& str);

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
}; // end of SimResult

// ******* Distribution *********//
class Distribution : public RWCollectable
{ RWDECLARE_COLLECTABLE(Distribution)
 public:
        RWCString Name,Type, Parameters, CmdLine;
        Distribution():Name(""),Type(""), Parameters(""){}
        Distribution(const RWCString& name, const RWCString& type="",
                        const RWCString& parameters="");
        ~Distribution(){};
        RWCString& GenCmdLine();
        virtual Boolean IsDefined();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
        int compareTo(const RWCollectable* c) const
        {       Distribution* b = (Distribution*)c;
                if(Name == b->Name) return 0;
                return Name>b->Name? 1: -1;
        }
        RWBoolean  isEqual(const RWCollectable* c) const
        {       const Distribution* b = (Distribution*) c;
                return Name == b->Name;
        }
};

// ****** Assignment *********//
class Assignment : public RWCollectable
{
 RWDECLARE_COLLECTABLE(Assignment)
 public:
        RWCString Branch;
        float Prob;
        Assignment():Branch(""){Prob = 0.0;}
        Assignment(const RWCString& branch, float prob=0.0);
        ~Assignment(){};
         int compareTo(const RWCollectable* c) const
        {       Assignment* b = (Assignment*)c;
                if(Branch == b->Branch) return 0;
                return Branch>b->Branch? 1: -1;
        }
        RWBoolean       isEqual(const RWCollectable* c) const
        {       const Assignment* b = (Assignment*) c;
```

```
                return Branch == b->Branch;
        }

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
        //Collectable methods
}; // end of Distribution

// ****** SimRun ******
class SimRun : public RWCollectable
{ RWDECLARE_COLLECTABLE(SimRun)
 protected:
        DFD* MyDFD;
        float Run;
        float Warmup;
        float Length;
        RWCString CmdLine;
public:
        SimRun(){MyDFD = NULL; Run = 0.0; Warmup = 0.0; Length = 0.0;};
        SimRun(DFD* dfd, float r=1, float w=120, float l=480);
        ~SimRun(){};
        Boolean IsDefined();
        DFD* GetMyDFD(){return MyDFD;}
        virtual RWCString& GenCmdLine();
        float GetRun() {return Run;}
        void SetRun(float r) {Run = r;}
        float GetWarmup(){return Warmup;}
        void SetWarmup (float w){ Warmup = w;}
        float GetLength(){ return Length;}
        void SetLength(float l){ Length = l;}

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
}; // end of SimRun

// ******** SimInfor ********
class SimInfor: public RWCollectable
{ RWDECLARE_COLLECTABLE(SimInfor)
  protected:
        RWCString Name, CmdLine;
        DFDObject*   dfdObj;
        int         Type;
        SimResult *  Result;
public:
        SimInfor():Name(""){Result = NULL; dfdObj = NULL;Type = 0;};
        SimInfor(DFDObject* obj, int type);
        ~SimInfor();
```

```
        int GetType(){return Type;}
        virtual RWCString & GetName();
        virtual RWCString& GenCmdLine();
        virtual Boolean IsDefined() {return TRUE;}
        SimResult* GetResult(){return Result;}
        int compareTo(const RWCollectable* c) const
        {       SimInfor* b = (SimInfor*)c;
                if(Name == b->GetName()) return 0;
                return Name>b->GetName()? 1: -1;
        }
        RWBoolean isEqual(const RWCollectable* c) const
        {       SimInfor* b = (SimInfor*) c;
                return Name == b->GetName();
        }

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
}; // end of SimInfor

// ****** SimAssign *******//
class SimAssign: public SimInfor
{ RWDECLARE_COLLECTABLE(SimAssign)
 protected:
        DbList* Assign;
 public:
        SimAssign(){Assign = NULL;};
        SimAssign(DFDObject* obj, int type);
        ~SimAssign();

        DbList * GetAssignList(){return Assign;}
        void SetAssignList(DbList* list);
        Assignment * GetAssign(int i)
        {       if(i<0) return NULL;
                return (Assignment*)Assign->at(i);
        }
        Assignment * GetAssign(const RWCString& a);
        float GetAssignProb(const RWCString& a);
        float GetAssignProb(Assignment* a);
        void SetAssignProb(float p, int i);
        void SetAssignProb(float p, const RWCString & a);
        void SetAssign(Assignment* a, int i);
        int GetBranches(){return Assign->entries();}

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
}; // end of SimAssign
```

```
// ******* SimResource *******//
class SimResource : public SimAssign
{ RWDECLARE_COLLECTABLE(SimResource)
 protected:
        RWCString Priority_Mode;
 public:
        SimResource():Priority_Mode(""){};
        SimResource(DFDObject* obj);
        SimResource(const RWCString& name);
        ~SimResource();
        virtual RWCString& GetName();
        virtual RWCString& GenCmdLine();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
}; // end of SimResource

// ****** SimSrc ********//
class SimSrc : public SimAssign
{ RWDECLARE_COLLECTABLE(SimSrc)
  protected:
        Distribution* Distr;
        float Start;
        float End;
        RWCString Event_Mode;
        RWCString Assign_Mode;  // set by a DFDObject depends on connections
  public:
        SimSrc(){};
        SimSrc(DFDObject* obj);
        ~SimSrc();

        Distribution* GetDistr() { return Distr;}
        void SetDistr(Distribution* d)
        {       if(Distr!=NULL)
                {       Distr->Name = d->Name;
                        Distr->Type = d->Type;
                        Distr->Parameters = d->Parameters;
                        delete d;
                }
                else    Distr=d;
        }
        float GetStart(){ return Start;}
        void SetStart(float s){ Start = s;}
        float GetEnd(){ return End;}
        void SetEnd(float e){ End = e;}
        RWCString& GetEvent_Mode(){ return Event_Mode;}
        RWCString& GetAssign_Mode()
        {       if (GetBranches() >1) Assign_Mode = "PROB";
                else Assign_Mode = "DET";
                return Assign_Mode;
```

```
        }
        virtual Boolean IsDefined();
        virtual RWCString & GetName();
        virtual RWCString& GenCmdLine();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
}; // end of SimSrc

// ********* SimSink ********//
class SimSink : public SimInfor
{ RWDECLARE_COLLECTABLE(SimSink)
 public:
        SimSink(){};
        SimSink(DFDObject* obj);
        ~SimSink(){};
        virtual RWCString & GetName();
        virtual RWCString& GenCmdLine();
}; //end of SimSink

// ******* SimFlow *******//
class SimFlow : public SimInfor
{ RWDECLARE_COLLECTABLE(SimFlow)
 protected:
        RWCString Source, Destine;
        float Prob;
 public:
        SimFlow(){Source = ""; Destine = ""; Prob = 0.0;}
        SimFlow(DFDObject* obj);
        ~SimFlow(){};
        virtual RWCString & GetName();
        virtual RWCString & GetSrcName();
        virtual RWCString & GetDestName();
        float GetProb();
        void SetProb(float p);
        SimBranch* ConvertToBranch();
        SimBranch* ConvertToBranch(SimFlow* f);
                //multiplex two flows
}; // end of SimFlow

//********** SimBranch **********//
class SimBranch : public RWCollectable
{ RWDECLARE_COLLECTABLE(SimBranch)
  protected:
        RWCString Source, Destine, CmdLine;
        float Prob;
  public:
        SimBranch(){Source = ""; Destine = ""; Prob = 0.0;}
        SimBranch(const RWCString& src, const RWCString& dest, float p=1.0);
        ~SimBranch(){}
```

```
        virtual RWCString& GenCmdLine();
        RWCString& GetSource(){return Source;}
        RWCString& GetDestine(){return Destine;}
        float GetProb() {return Prob;}
        void SetSource(const RWCString& s) {Source =s;}
        void SetDestine(const RWCString& d) {Destine = d;}
        void SetProb(float p){Prob = p;}
        int compareTo(const RWCollectable* c) const
        {       SimBranch* b = (SimBranch*)c;
                if(Source == b->GetSource()) return 0;
                return Source>b->GetSource()? 1: -1;
        }
        RWBoolean  isEqual(const RWCollectable* c) const
        {       SimBranch* b = (SimBranch*) c;
                return (Source == b->GetSource()&&Destine==b->GetDestine());
        }
}; // end of SimBranch

//******** SimQ *******//
class SimQ : public SimInfor
{ RWDECLARE_COLLECTABLE(SimQ)
 protected:
        RWCString Priority_Mode;
        RWCString Resource;
 public:
        SimQ():Priority_Mode(""), Resource(""){};
        SimQ(DFDObject* obj);
        ~SimQ(){}
        RWCString& GetP_Mode(){return Priority_Mode;}
        RWCString& GetResource(){return Resource;}
        void SetResource(const RWCString  r) {Resource=r;}
        virtual RWCString & GetName();
        virtual RWCString& GenCmdLine();

        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
}; // end of SimQ

// ****** SimAct ********
class SimAct : public SimAssign
{  RWDECLARE_COLLECTABLE(SimAct)
 protected:
        RWCString Resource;
        Distribution * Distr;
        RWCString Assign_Mode;
 public:
        SimAct():Assign_Mode(""), Resource("") {Distr = NULL;};
        SimAct(DFDObject* obj);
        ~SimAct();
        virtual Boolean IsDefined();
```

```
        Distribution* GetDistr() { return Distr;}
        void SetDistr(Distribution* d)
        {       if(Distr!=NULL)
                {       Distr->Name = d->Name;
                        Distr->Type = d->Type;
                        Distr->Parameters = d->Parameters;
                        delete d;
                }
                else    Distr=d;
        }
        RWCString& GetQ();
        RWCString& GetResource(){return Resource;}
        void SetResource(SimResource* r)
        {       if(r!=NULL)
                Resource = r->GetName();
        }
        void SetResource(const RWCString& r)  { Resource=r;}
        RWCString& GetAssign_Mode()
        {       if (GetBranches() >1) Assign_Mode = "PROB";
                else Assign_Mode = "DET";
                return Assign_Mode;
        }

        virtual RWCString & GetName();
        virtual RWCString& GenCmdLine();
        // Inherited from class "RWCollectable":
        unsigned        binaryStoreSize() const;
        void            restoreGuts(RWvistream&);
        void            saveGuts(RWvostream&) const;
}; // end of SimAct
#endif
```

E.3. Simulation subsystem

1) The main window

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  SimInfor
// Main window of simulation subsystem
//        SuperClass: RWCollectable
//        File:     siminfor.h
//        Author: Jackson He 02/94
//        Language C++

#ifndef _YANSLWindow
#define _YANSLWindow

#define CM_Run              100

#include <owl.h>
#include <filewnd.h>
#include "mydde.h"

_CLASSDEF(YANSLApp)
_CLASSDEF(YANSLWindow)
_CLASSDEF(SimModelGen)

//****  Declare YANSLApp, a TApplication descendant *****//
class _CLASSTYPE YANSLApp : public TApplication
{
public:
        YANSLApp(LPSTR name, HINSTANCE hInstance,
                HINSTANCE hPrevInstance, LPSTR lpCmd, int nCmdShow)
            : TApplication(name, hInstance,  hPrevInstance, lpCmd, nCmdShow) {};
        virtual void InitMainWindow();
        virtual void InitInstance();
}; // end of YANSLApp

// **** Declare YANSLWindow, a TMDIFrame descendant ****//
class _CLASSTYPE YANSLWindow : public MyDDEWindow
{
protected:
        virtual void SetupWindow();
        virtual void NewFile(RTMessage Msg) = [CM_FIRST + CM_MDIFILENEW];
        virtual void OpenFile(RTMessage Msg) = [CM_FIRST + CM_MDIFILEOPEN];
        virtual void CMRun(RTMessage Msg) = [CM_FIRST + CM_Run];
        void ShowResult();
        PTFileWindow Model, Result;
        SimModelGen * Simulator;
        void GetWindowClass( WNDCLASS& WndClass );
public:
        YANSLWindow(LPSTR ATitle, LPSTR MenuName);
        ~YANSLWindow();
}; // end of YANSLWindow
#endif
```

2) The model generator

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  ModelGen
// Simulation model generator
//        SuperClass: RWCollectable
//        File:     siminfor.h
//        Author: Jackson He 02/94
//        Language C++

#ifndef _SimModelGen
#define _SimModelGen
#include <owl.h>
#include <filewnd.h>
#include <mdi.h>
#include <string.h>
#include <io.h>
#include "rw/cstring.h"
#include "yanslwin.h"

#define ModelSize  2500
#define BlockSize  2500
#define LineLen    150
#define KeyLen     35
#define NumP       6 // number of parameters in a command line

_CLASSDEF(SimModelGen)
_CLASSDEF(Simulation)
_CLASSDEF(Command)
_CLASSDEF(Random)

// **** Object to hold a simulation command ***
class Command
{
 private:
        RWCString Name;
        RWCString Type;
 public:
        RWCString Opr, p1, p2, p3, p4, p5;
        void * CmdObj;
        Command* Next;
        Command(const RWCString& name, const RWCString& type);
        ~Command(){}
        const RWCString& GetName() const;
        const RWCString& GetType() const;
        Boolean operator == (const Command& c) const;
        Boolean operator <(const Command& p) const;
}; // end of Command
```

```
// ****** SimModelGen *****//
class SimModelGen
{
 public:
        SimModelGen(PYANSLWindow parent);
        ~SimModelGen();
        BOOL RunSimModel();
 protected:
        PYANSLWindow Parent;
        Command* CmdListHead;
        void ClearCmdList(); // string operations
        int FindNext(char* buffer, char* str, int start); //find next location of str in buffer
        int GetNextLine(char buffer[], char temp[], int start); // return next line of buffer to temp
        int GetNextBlock(char buffer[], char temp[], int start); // return a block of buffer divided by {}
        void De_Space(char*); // get rid of redundant spaces
        // list operations
        Command* FindACommand(char* str, char* type);
        Command* FindACommand(Command* cmd);
        Command* AppendACommand(Command* cmd);
        // parser operations
        Simulation*  ParseRun(char* str);
        int ParseDistributions(char* str);
        int ParseResources(char* str);
        int ParseSources(char* str);
        int ParseQueues(char* str);
        int ParseActivities(char* str);
        int ParseSinks(char* str);
        int ParseBranches(char* str);
        void ParseCommand(char* line, char* p1, char* p2, char* p3, char* p4, char* p5, char* p6);
        Random* GetDistribution(char*);
        void* GetCommandObj(char*);
        Command* GetCommand(char*);
}; // end of SimModelGen
#endif
```

E.4. DDE Data interface

1. DDE Window

```
// Copyright (C) 1993 by University of Hawaii
// Hyper Analysis Toolkit (R)  MyDDE
// DDE data interface
//        SuperClass:TMDIFrame
//        File:     mydde.h
//        Author: Jackson He 02/94
//        Language C++

#ifndef _MyDDE
#define _MyDDE

#include <owl.h>
#include <inputdia.h>
#include <ddeml.h>
#include <stdio.h>
#include <string.h>
#include <mdi.h>
#include "ddedef.h"
#include "rw/cstring.h"
#include "apphdl.h"

#define BufLen 2000
#define NumDDE 2
#define NumItem 1

_CLASSDEF(MyDDEWindow)
class MyDDEWindow : public TMDIFrame
{
 friend AppHandle;
 friend SimAppHandle;
 protected:
        // client agent
        AppHandle* Appl;
        MyDDEWindow( LPSTR, LPSTR );
        virtual ~MyDDEWindow();
        virtual void SetupWindow();
        virtual void InitialApp();
        virtual void SetupClient();
        virtual void CloseClient();
        Boolean ConnectToServer(int clientId);
        virtual void SendRequest(int clientId, int reqId);
        virtual void SendPoke(int clientId, int reqId, char* pokeMsg);
        Boolean StartAdvice(int clientId, int reqId, DWORD*);
        Boolean StopAdvice(int clientId, int reqId, DWORD*);
        virtual void WMInitMenu( RTMessage ) = [WM_FIRST + WM_INITMENU];
        virtual void CMExit( RTMessage ) = [CM_FIRST + CM_EXIT];
        virtual void CMConnect1( RTMessage ) = [CM_FIRST + CM_Connect1];
        virtual void CMConnect2( RTMessage ) = [CM_FIRST + CM_Connect2];
```

```
        virtual void CMRequest1( RTMessage ) = [CM_FIRST + CM_Request1];
        virtual void CMRequest2( RTMessage ) = [CM_FIRST + CM_Request2];
        virtual void CMPoke1( RTMessage ) = [CM_FIRST + CM_Poke1];
        virtual void CMPoke2( RTMessage ) = [CM_FIRST + CM_Poke2];
        virtual void CMDisconnect1( RTMessage ) = [CM_FIRST + CM_Disconnect1];
        virtual void CMDisconnect2( RTMessage ) = [CM_FIRST + CM_Disconnect2];
        virtual void CMUHelpAbout( RTMessage ) = [CM_FIRST + CM_U_HELPABOUT];
        virtual void ReceivedData( HDDEDATA, int);
        static HDDEDATA FAR PASCAL _export ClientCallBack( WORD, WORD,
                HCONV, HSZ, HSZ, HDDEDATA, DWORD, DWORD );

        DWORD idInstClient;
        HCONV hConvClient[NumDDE];
        BOOL tfLoopClient;
        HSZ hszServiceClt[NumDDE], hszTopicClt[NumDDE], hszItemClt[NumDDE];
        FARPROC lpClientCallBack;
        int IsMyClientConv(HCONV hConv);

        //server agent
        virtual void CloseServer();
        virtual void SetupServer();
        virtual BOOL MatchTopicAndService( HSZ, HSZ );
        virtual int MatchTopicAndItem( HSZ, HSZ );
        virtual HDDEDATA WildConnect( HSZ, HSZ, WORD );
        virtual HDDEDATA DataRequested( WORD, int );
        virtual void UpdateData();
        static HDDEDATA FAR PASCAL _export ServerCallBack( WORD, WORD,
                HCONV, HSZ, HSZ, HDDEDATA, DWORD, DWORD );

        DWORD idInstServer;
        HCONV hConvServer;
        BOOL tfLoopServer;
        HSZ hszServiceSvr, hszTopicSvr, hszItemSvr;
        FARPROC lpServerCallBack;
}; // end of my DDE
#endif
```

2. Application handlers

```
#ifndef _AppHandle
#define _AppHanle
#include "rw/cstring.h"

#define NumItem 1
#define NumDDE  2
_CLASSDEF(MyDDEWindow)
_CLASSDEF(YANSLWindow)

//****** AppHandler******//
class AppHandle
{
```

```
public:
        AppHandle(MyDDEWindow* interface) : NullStr("") { ParentWin = interface;}

        // server methods
        void SetServerName(const RWCString& sname) { SName = sname;}
        RWCString & GetServerName() { return  SName;}
        void SetServerService(const RWCString& sserve) { SService = sserve;}
        RWCString & GetServerService() { return SService;}
        void SetServerTopic(const RWCString& stopic) { STopic = stopic;}
        RWCString & GetServerTopic() { return STopic;}
        void SetServerItem(const RWCString& sitem, int j)
           { if(j>=0 && j<NumItem) SItem[j] = sitem;}
        RWCString & GetServerItem(int j)
        {       if(j>=0 && j<NumItem) return SItem[j];
                else return NullStr;
        } // set and get the j-th item
        virtual RWCString& PrepareData(int i, const RWCString& request = "");
        virtual void PokeMsgHandler(char *);

        // Client methods
        void SetClientName(const RWCString& cname) { CName = cname;}
        RWCString & GetClientName() { return CName;}
        void SetClientService(const RWCString& cserve, int i)
                { if(i>=0 && i<NumDDE) CService[i] = cserve;}
        RWCString & GetClientService(int i)
        {       if(i>=0 && i<NumDDE) return CService[i];
                else return NullStr;
        }
        void SetClientTopic(const RWCString& ctopic, int i)
                { if(i>=0 && i<NumDDE) CTopic[i] = ctopic;}
        RWCString & GetClientTopic(int i)
        {       if(i>=0 && i<NumDDE) return CTopic[i];
                else return NullStr;
        }
        void SetClientItem(const RWCString& citem, int i, int j)
        {       if(i>=0 && i<=NumDDE && j>=0 && j<NumItem)
                 CItem[i][j] = citem;
        }
        RWCString & GetClientItem(int i, int j)
        {       if(i>=0 && i<=NumDDE && j>=0 && j<NumItem)
                return CItem[i][j];
                else return NullStr;
        }
        void SetClientBuffer(const RWCString& cbuf, int i)
                { if(i>=0 && i<NumDDE) CBuffer[i] = cbuf;}
        RWCString & GetClientBuffer(int i)
        {       if(i>=0 && i<NumDDE) return CBuffer[i];
                else return NullStr;
        } // set and get the j-th item of i-th client
        virtual RWCString& ReceiveData(int i);
        RWCString& GetDataBuffer(){return DataBuffer;}
        void SetDataBuffer(const RWCString& s){DataBuffer = s;}
```

```
protected:
        MyDDEWindow* ParentWin;
        RWCString NullStr;
        RWCString DataBuffer;
        // Server attributes
        RWCString SName;
        RWCString SService;
        RWCString STopic;
        RWCString SItem[NumItem];
        // Client attributes
        RWCString CName;
        RWCString CService[NumDDE];
        RWCString CTopic[NumDDE];
        RWCString CItem[NumDDE][NumItem];
        RWCString CBuffer[NumDDE];
}; // end of AppHandle



// ****** UI (User Interface) Handle *******//
class UIAppHandle:public AppHandle
{
 public:
        UIAppHandle(MyDDEWindow* myDad):AppHandle(myDad){}
        virtual RWCString& PrepareData(int i, const RWCString& request = "");
        virtual void PokeMsgHandler(char *);
        virtual RWCString& ReceiveData(int i);
        virtual void ConnectToSim();
        virtual void ConnectToES();
        virtual RWCString& RequestFromSim();
        virtual RWCString& RequestFromES();
        virtual void PokeToSim(char* msg);
        virtual void PokeToES(char* msg);
        RWCString& PrepareESData(const RWCString& request);
        RWCString& PrepareSimData(const RWCString& request);
        Boolean IsConnected(int i);
}; // end of UIAppHandle

// ***** SimAppHandle ******//
class SimAppHandle:public AppHandle
{
 public:
        SimAppHandle(YANSLWindow* myDad);
        virtual RWCString& PrepareData(int i, const RWCString& request = "");
        virtual void PokeMsgHandler(char *);
        virtual RWCString& ReceiveData(int i);
        virtual void ConnectToUI();
        virtual void ConnectToES();
        virtual RWCString& RequestFromUI();
        virtual RWCString& RequestFromES();
        virtual void PokeToUI(char* msg);
        virtual void PokeToEs(char* msg);
```

```
        RWCString& PrepareESData(const RWCString& request);
        RWCString& PrepareUIData(const RWCString& request);
        Boolean IsConnected(int i);
};
#endif
```

BIBLIOGRAPHY

[Abowd 89] Abowd, G., Bowen, J., Dix. A., Harrison, M., and Took, R. "User Interface Languages: A Survey of Existing Methods", Programming Research Group Report PRG-TR-5-89, Oxford University Computing Laboratory, October, 1989

[Ahituv 90] Ahituv, N. and Neumann, S. "Principles of Information Systems for Management, 3rd edition", C. Brown Publishers, 1990

[August 91] August, J. H. "Joint Application Design", Yourdon Press, 1992

[Balci 87] Balci, O. and Nance, R. "Simulation Support: prototyping the automation-based paradigm", Proceedings of the 1987 Winter Simulation Conference, Dec. 1987, pp 495-502

[Balci 92] Balci, O. and Nance, R. "The Simulation Model Environment: An Overview", Proceedings of the 1992 Winter Simulation Conference, Dec. 1992, 726-736

[Bell 87] Bell, P. C. and O'Keefe R. M. "Visual Interactive Simulation - History, Recent Developments, and Major Issues", Simulation, Vol. 49, No. 3, March 1987, pp 109-116

[Bell 91] Bell, P. C. "Visual Interactive Modeling: The Past, the Present, and the prospects", European Journal of Operational Research, Vol. 54, 1991, pp 274-286

[Bischak 91] Bischak, D. P. and Roberts, S. D. "Object-Oriented Simulation", Proceedings of the Winter Simulation Conference, 1991, pp 194-203

[Brooks 87] Brooks, F. P. "No silver bullet: essence and accidents of software engineering", IEEE Computer, April 1987

[Carando 89] Carando, P. "SHADOW: Fusing Hypertext with AI", IEEE Expert, Winter 1989, pp65-78

[Card 91] Card, S.K., Robertson, G.G., and Mackinlay, J.D. "The Information Visualizer: An Information Workspace" Proceedings of CHI'91 Human Factors in Computing Systems, New Orleans, LA, 1991, pp 181-188

[Carmel 92] Carmel, E., G., Joey F. and Nunamaker, J. F. Jr. "Supporting joint application development (JAD) and electronic meeting system: moving the CASE concept into new areas of software development", Proceedings of the HICSS, Maui 1992, vol. 3

[Chen 76] Chen, P. "Entity-relation Approach", ACM Trans. Database System Vol. 1. 1, 1976

[Chen 89] Chen, M., Nunamaker J. F. Jr. and Weber, E. S. "Computer-aided software engineering: present status and future directions", Data Base, Spring, 1989

[Chen 92a] Chen, M. and Norman, R. J. "Integrated CASE: adoption, implementation and impacts", Proceedings of the HICSS, Maui 1992, vol. 3

[Chen 92b] Chen, M., Norman, R. J. "A Frame for Integrated CASE", IEEE Computer, Mar. 1992

[Coad 90] Coad, P. and Yourdon, E. "Object-oriented Analysis", Prentice Hall Inc. 1990

[Coad 91] Coad P. and Yourdon, E. "Object-oriented Design", Prentice Hall Inc. 1991

[Cobb 90] Cobb, R. H. and Mills, H. D. "Engineering Software under Statistical Quality Control", IEEE Software, Nov. 1990, pp 44-54

[Conklin 87] Conklin, J. "Hypertext: An Introduction and Survey", IEEE Computer, Vol. 20 No. 9, 1987 pp 17-37

[Cybulski 92] Cybulski, J. L. and Reed, K. "A hypertext based software engineering environment", IEEE Software, Mar. 1992, pp 62-68

[Dupuy 90] Dupuy, A., Schwartz, J., Yemimi, Y. and Bacon, D. "NEST: a network simulation and prototyping testbed", Communications of the ACM, Vol. 33, No. 10, October 1990

[Eddins 90] Eddins, W., Crosslin, R. Sutherland, D. E. "Using Modeling and Simulation in the Analysis and Design of Information Systems", Proceedings of International Working Conference on Dynamic Modeling of Information Systems, April, 1990

[Eich 89] Eich, M, Fan, C., Sun, W. and Rafiqi, S. "A methodology for simulation of database systems", Simulation, June 1989, 241-254

[Fischer 89] Fischer, G. "Human-computer interface software: Lessons Learned, Challenges ahead", IEEE Software, Jan. 1989, pp 44-52

[Fox 89] Fox, M. S., Husain, N., McRoberts, M. and Reddy, Y. V. "Knowledge-Based Simulation: An Artificial Intelligence Approach to System Modeling and Automating the Simulation Life Cycle", *Artificial Intelligence, Simulation and Modeling,* John Wiley&Sons Inc., 1989, pp447-485

[Frankel 89] Frankel, V. L. and Balci, O. "An on-line assistance system for the simulation model development environment", International Journal of Man-Machine Studies, Vol. 31, pp699-716

[Galitz 93] Galitz, W. O. "User-Interface Screen Design", QED Information Sciences, Inc. 1993

[Gane 79] Gane, C. and Sarson, T. "Structured systems analysis: tools and techniques", Prentice Hall Inc., 1979

[Gane 90] Gane, C. "CASE: the methodologies, the products and the future", Prentice Hall Inc., 1990

[Gerlach 91] Gerlach, J. H. and Kuo, F. Y. "Understanding Human-Computer Interaction for Information System Design", MIS Quarterly, Dec. 1991, pp527-549

[Gould 85] Gould, J. D. and Lewis, C. "Designing for usability: key principles and what the users think", Communications of the ACM, Vol. 28, No. 3, March 1985, pp 300-311

[Gore 90] Gore, A. "QASE to configure huge systems", MACWEEK, Nov. 13, 1990

[Graber 90] Graber, A., Ulrich, H. and Bolay, F. "Object-Oriented General Purpose Simulator Based on Interactive Petri Nets", Proceedings of 1990 Summer Simulation Conference 1990, pp 843-847

[Griggs 89] Griggs K. "GDI: (Goal Directed Interface): An intelligent, Iconic, object-oriented interface for office systems", Ph.D. Dissertation, University of Arizona, 1989

[Gronbaek 94] Gronbaek K. and Trigg R. H. "Design issues for a DEXTER-based hypertext system", Communications of the ACM February 1994, pp40-49

[Hartson 89] Hartson, H. R. and Hix, D. "Human-computer Interface Development: Concepts and Systems for its Management", ACM Computer Survey, Vol. 21 No.1, Mar. 1989, pp 5-85

[Halasz 94] Halasz, F. and Schwartz "The Dexter hypertext reference model", Communications of the ACM, February 1994, pp 30-39

[He 94a] He, j. and Griggs, K. "A Tool for Hypertext-based Systems Analysis and Dynamic Evaluation". Proceeding of 27th HICSS, Maui, Hawaii, Jan. 4-7, 1994

[He 94b] He, J. Wild, R. and Griggs K. "An architecture to support reverse simulation", The 1994 Summer Conference on Computer Simulation

[Henriksen 83] Henriksen, J. O. "The Integrated Simulation Environment", Operation Research Vol. 31, No.6, Nov.-Dec. 1983 pp 1053-1072

[Hill 87] Hill, T. R. and Roberts, S. D. "A prototype knowledge-based simulation support system", Simulation, April 1987

[Horton 90] Horton, W. K. "Design and writing on-line documentation: help file to hypertext", John Wiley & Sons, New York, 1990

[Hurrion 91] Hurrion, R. D. "Intelligent Visual Interactive Modeling", European Journal of Operational Research, Vol. 54, 1991, pp 349-356

[Ignizio 91] Ignizio, J. P. "An Introduction to expert systems", McGraw-Hill, Inc., 1991

[Ives 83] Ives, B, Olson, M. and Baroudi, J. "The Measurement of user information satisfaction", Communications of the ACM, Vol. 26, No. 10, pp 785-793, Oct. 1983

[Ives 84] Ives, B and Olson, M "User involvement and MIS success: a review of research", Management Science, 30, pp 586-603

[Joines 92] Joines, J. A., Powell, K. A. Jr and Roberts, S. D. "Object-oriented modeling and simulation with C++", Proceedings of the 1992 Winter Simulation Conference

[Keller 87] Keller, R. "Expert System Technology", Yourdon Press, A Prentice Hall Company, 1987

[Kimbler 88] Kimbler, D. L. and Watford, B. A. "Simulation program generation: A functional perspective", Proceedings of the SCS Multiconference on AI and Simulation, Feb. 3-5 1988, San Diego, CA

[Kreutzer 90] Kreutzer, W. "The modeler's Assistant - a first step toward integration of knowledge bases and modeling systems", Proceedings of Summer Computer Simulation Conference, July 1990, pp 874-879

[Kwanjai 92] Kwanjai, N. K. and Wild, R. H. "A Recursive Expert System to Facilitate Simulation Experimentation: Discussion and Prospects for a Reverse Simulation Technique", Proceedings of Winter Simulation Conference, 1992, Washington

[Lantz 87] Lantz, K. E. "The Prototyping Methodology", Prentice Hall Inc. 1987

[Law 91] Law, A. M. and Kelton, W. D. "Simulation modeling and analysis, 2nd edition", McGraw-Hill Inc., 1991

[Liou 93] Liou, Y. I. and Chen, M. "Integrating Group Supporting Systems, Joint Application Development, and Computer-Aided Software Engineering for Requirement Specification", Proceedings of HICSS, Maui, Hawaii 1993

[Lomow 90] Lomow, G. and Baezner, D. "A Tutorial Introduction to Object-Oriented Simulation and Sim++", Proceedings of the Winter Simulation Conference, 1990, pp 149-153

[Marcus 91] Marcus, A. "User-interface development in the nineties", IEEE Computer, Sep. 1991

[Martin 88] Martin, J. and McClure, C. "Structured Techniques: The Basis for CASE", Prentice Hall Inc., 1988

[Martin 90a] Martin, J. "Use of Automated tools Crucial to RAD life cycle success", PC Week, January 15, 1990

[Martin 90b] Martin, J. "Hyperdocuments and how to create them", Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[McAleese 89] McAleese, R. "Hypertext: theory into practice", Ablext Publishing Corporation, 1989

[McAleese 90] McAleese, R. "Hypertext: state of the art", Ablext Publishing Corporation, 1990

[McClure 89] McClure, C. "CASE is software automation", Prentice Hall Inc., 1989

[Mellichamp 89] Mellichamp, J. M. and Park, Y. H. "A statistic expert system for simulation analysis", Simulation, April 1989, pp134-139

[Mittermeir 90] Mittermeir, R. T. and Rossak, N. "Reusability", Chapter 7 of "Modern software Engineering", VNR, New York, pp 205 - 235

[Myers 89] Myers, B. A. "User-Interface Tools: Introduction and Survey", IEEE Software, Jan. 1989, pp 15-23

[Mynatt 92] Mynatt, B. T. Leventhal, L. M., Instone, K. Farhat, J., Rohlman, D. "Hypertext or Book: Which is better for Answering Questions?", Proceedings of CHI'92, Mar. 1992, pp 19-25

[Norman 86] Norman, D. A. and Draper, S. W. "User-Centered System Design", Hillsale, NJ. Lawrence Erlbaum, 1986

[Oinas-Kukkonen 93] Oinas-Kukkonen, H. "Intermediary Hypertext Systems in CASE Environments", Research Papers Series A16, Department of Information Processing Science, University of Oulu, Finland

[O'Keefe 86] O'Keefe, R. "Simulation and Expert System - a taxonomy and some examples", Simulation, Jan. 1986 pp10-15

[O'Keefe 87] O'Keefe, R. M. "What is Visual Interactive Simulation?", Proceedings of the 1987 Winter Simulation Conference, 1987, pp 461-464

[O'Keefe 89] O'Keefe, R. "The Role of Artificial Intelligence in Discrete-Event Simulation", *Artificial Intelligence, Simulation and Modeling*, John Wiley&Sons Inc., 1989, pp359-379

[O'Keefe 92] O'Keefe, R. M. and Bell, P. C. "Findings from Behavioral Research in Visual Interactive Simulation", Proceedings of the 1992 Summer Simulation Conference, July 1992, pp 751-755

[Oman 90] Oman, P. W. "CASE analysis and design tools", IEEE Software, May 1990, pp37-43

[Park 90] Park, Y. H. and Mellichap, J. M. "A statistical expert system for simulation analysis", Proceedings of Summer Simulation Conference, 1990

[Pleas 94] Pleas, K. "OLE 2.0: Putting the pieces together", Visual Basic Programmer's Journal, March / April 1994

[Rao 88] Rao, M. J. and Sargent, R. G. "An experimental advisory system for operational validity", Proceedings of the SCS Multiconference on AI and Simulation, Feb. 3-5 1988, San Diego, CA

[Royce 70] Royce, W. W. "Managing the development of large software system", Proceedings of WESTCON 1970, CA, USA

[Sol 91] Sol, H. G. "Dynamics in information system", Proceedings of Dynamic Modeling of Information System II, 1991

[Sommerville 89] Sommerville, I. "Software Engineering, 3rd edition", Addison-Wesley Publishing Company, 1989

[Sprague 82] Sprague, R. H. and Carlson, E. D. "Building Effective Decision Support Systems", Prentice Hall, 1982

[Strong 90] Strong, B. "Requirements for database support in CASE", from "Modern Software Engineering", edited by P. Ng and R. Yeh, Van Nostrand Reinhold, 1990

[Taylor 88] Taylor, R. P. and Hurrion, R. D. "An expert advisor for simulation experimental design and analysis", Proceedings of the SCS Multiconference on AI and Simulation, Feb. 3-5 1988, San Diego, CA

[Towner 89] Towner, L. E. "CASE: Concept and Implementation", McGraw-Hill Book Company, Inc. 1989

[Vujosevic 90] Vujosevic, R. "Object-Oriented Visual Interactive Simulation", Proceedings of the Winter Simulation Conference, 1990, pp 490-498

[Warren 91] Warren, J. R. and Stott, J. W. "CASE/Simulation: making performance evaluation a normal part of information system development", Proceedings Dynamic Modeling of Information System II, 1991

[Warren 92] Warren, J. R., Stott, J. W. and Norcio, A. F. "Stochastic simulation of information system design from data flow diagrams", Journal of Systems Software, May 1992

[Warren 93] Warren, J. R. and Canfield, G.C. "Information systems performance evaluation: a study of the relationship between decision accuracy of systems analysis and design simulation usage", Technical Report #CSIS-93-005, Department of Computer Science and Information Systems, The American University

[Whitten 89] Whitten, J., Bentley, L. and Barlow, V. "Systems analysis & design methods, 2nd edition", Richard D. Irwin, Inc. 1989

[Widman 89] Widman, L. E. "Artificial Intelligence, Simulation, and Modeling: A Critical Survey", *Artificial Intelligence, Simulation and Modeling,* John Wiley&Sons Inc., 1989, pp1-44

[Wild 91a] Wild, R. H. and Griggs, K. A. "Improving the Quality of Information Systems Analysis and Design Through Simulation Modeling", Proceedings of ISAGA 1991

[Wild 91b] Wild, R. H. and Pignatiello, J. J. "An Expert System-based Reverse Simulation Technique", Proceedings of 1991 Summer Simulation Conference, July 1991, pp 352-357

[Wild 93] Wild, R. H. and Griggs, K. A. "Design robust information systems", Proceedings of the 26th Hawaii International Conference on System Science, Vol. IV, pp 419-428, Jan 4-7, 1993

[Wright 83] Wright, P and Lickorish, A. "Proof-reading texts on screen and paper", Behavior and information Technology, Vol. 2 No. 3, 1983, pp227-235

[Wu 90] Wu, X. "On Expert Simulation System", Proceedings of 1990 Summer Simulation Conference, July 1990, pp 715-720

[Yourdon 79] Yourdon, E. "Structured walkthroughs, 2rd edition", Prentice Hall 1979

[Yourdon 89] Yourdon, E. "Managing the Structured Techniques, 4th edition", Prentice Hall Inc., 1989

[Yourdon 92] Yourdon, E. "The Decline and Fall of the American Programmers", Prentice Hall Inc., 1992

[Zhang 89] Zhang, Q. and Zeigler, B. P. "The system entity structure: knowledge representation for simulation modeling and design", *Artificial Intelligence, Simulation and Modeling,* John Wiley&Sons Inc., 1989, pp47-73