NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

A MODEL AND DECISION SUPPORT MECHANISM FOR SOFTWARE REQUIREMENTS ENGINEERING

by

Osman Mohamed Ibrahim

September 1996

Thesis Advisor: Valdis Berzins

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1996	3. REPORT TYPE AND DATES COVERED Ph.D. Dissertation

4. TITLE AND SUBTITLE A MODEL AND DECISION SUPPORT MECHANISM FOR SOFTWARE REQUIREMENTS ENGINEERING	5.
6. AUTHOR(S) Osman Mohamed Ibrahim.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This dissertation introduces a formal model for requirements analysis and evolution and a decision support mechanism based on that model. Both the model and the decision support mechanism provide automated support for the early part of the prototyping process. The model is used to capture user reactions to the demonstrated behavior of a prototype and map these reactions into the model objects to be used in synthesizing a set of open issues to be resolved. The issues are resolved by examining and modifying requirements if necessary, and then propagating the change consequences down into the affected parts of system specification and implementations in a consistent and controlled manner.

This process is performed through a set of analysis and design activities controlled by the manager and aided by the decision support mechanism based on the formal model. This approach also provides support for maintaining design history and its rationale that can be used for implementing new needs or performing comparative studies to choose among alternatives.

A formalism is also developed that supports customers in choosing among available alternatives to requirements that satisfy their goals and meet other constraints.

14. SUBJECT TERMS Software requirements, analysis and evolution, automated decision support, formal models, design rationales, design databases, prototyping.	15. NUMBER OF PAGES 331
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

An improved decision support method based on this formalism supports individuals that represent different customer view points to reach a final decision that represents the combined view of the group.

A database is an important component of any decision support mechanism. This work also provides a conceptual design of an engineering database capable of representing and managing the process knowledge. This knowledge includes all information related to a software prototype design. The management of this information includes storing, retrieving, viewing, and controlling the design knowledge. The design of this engineering database is based on the object oriented paradigm. This paradigm provides the representation power to easily map our model objects and their relationships efficiently and naturally.

A new implementation model has also been developed that provides smooth and safe communication between the implementation language and the database manipulation language. The new implementation technique based on that model also allows the implementation language to directly access the database facilities. This access is done without going through intermediate layers of codes that must be implemented in another language. This is not possible without the new technique.

A MODEL AND DECISION SUPPORT MECHANISM FOR SOFTWARE REQUIREMENTS ENGINEERING

Osman Mohamed Ibrahim
COL, Egyptian AirForce
B.S., Military Technical College, Cairo, Egypt 1977
M.S.C.E., Institute of Statistical Studies and Research, Cairo University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1996

Author: Osman Ibrahim
Osman Mohamed Ibrahim.

Approved By:

[illegible]
Luqi
Professor of Computer Science

[illegible]
Man-Tak Shing
Professor of Computer Science

[illegible]
Herschel H. Loomis, Jr
Professor of Electrical & Computer Engineering

[illegible]
Qing Wang
Professor of Meteorology

Valdis Berzins
Valdis Berzins
Professor of Computer Science
Dissertation Supervisor

Approved by: [illegible]
T. Lewis, Chairman, Department of Computer Science

Approved by: Maurice D. Weir
Maurice D. Weir, Associate Provost for Instruction

ABSTRACT

The early portion of the software prototyping process is missing automatic support for many important activities that help the software manager and the design team members firm up requirements and control the system design and evolution to satisfy the customers' real needs. This dissertation introduces a formal model for requirements analysis and evolution and a decision support mechanism based on that model. Both the model and the decision support mechanism provide the missing support identified above. Within the framework of this model the support provided spans the whole life cycle of the software development process. The model is used to capture user reactions to the demonstrated behavior of a prototype and map these reactions into the model objects to be used in synthesizing a set of open issues to be resolved. The issues are resolved by examining and modifying requirements if necessary, and then propagating the change consequences down into the affected parts of system specification and implementations in a consistent and controlled manner.

This process is performed through a set of analysis and design activities controlled by the manager and aided by the decision support mechanism based on the formal model. This approach also provides support for maintaining design history and its rationale that can be used for implementing new needs or performing comparative studies to choose among alternatives.

A formalism is also developed that supports customers in choosing among available alternatives to requirements that satisfy their goals and meet other constraints. An improved decision support method based on this formalism supports individuals that represent different customer view points to reach a final decision that represents the combined view of the group.

A database is an important component of any decision support mechanism. This work also provides a conceptual design of an engineering database capable of representing and managing the process knowledge. This knowledge includes all information related to a software prototype design. The management of this information includes storing, retrieving, viewing, and controlling the design knowledge. The design of this engineering database is based on the object oriented paradigm. This paradigm provides the representation power to easily map our model objects and their relationships efficiently and naturally.

A new implementation model has also been developed that provides smooth and safe communication between the implementation language and the database manipulation language. The new implementation technique based on that model also allows the implementation language to directly access the database facilities. This access is done without going through intermediate layers of codes that must be implemented in another language. This is not possible without the new technique.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First of all, I would like to acknowledge the unfailing love, devotion, and unconditional support which I have received throughout this experience from my wife, Lobna. She has given up so much and expected so little in return. Next, I must thank my children Walaa, Mahmoud, and Hoda who kept calling me "Doc" a long time ago.

Secondly, I wish to express my deepest gratitude to Professor Valdis Berzins whose support, guidance, knowledge, and enthusiasm have been a constant inspiration to me. His patience and positive attitude were invaluable to this research. Thirdly, I am also deeply indebted to the remaining members of my committee. Professor Luqi for her support and valuable advice that always eased my tensions. Professor Shing for the insightful discussions we often had. Professor Loomis for his support, caring, and comments regarding my writing. Professor Qing for her comments and continuous encouragement.

I also wish to thank my fellow Ph. D. students from the EE Dept., Nabil Khalil and Khalid Shehata who formed together with me the long nights of company.

Finally and most importantly I thank you God for making this all possible.

I. INTRODUCTION

This dissertation defines a conceptual model and a decision support mechanism for requirements analysis and validation in the computer-aided prototyping environment of software design and development. Traditional models of requirements analysis do not use the result of the requirements analysis process to provide automated decision support for the system design process. In our approach changes in requirements automatically expose the affected parts of the system design and implementation for use in automated project planning and configuration control. We also provide automated support for customers to pose their criticisms, discuss, and decide on the available alternatives to fix errors. The final outcome of this deliberation process is a change request that reflects the justified view point of the customer. Proposed plans for carrying out the work required by the change request are automatically generated for managers to review, adjust if necessary, and then approve. Our decision support mechanism also records and reasons with the design rationale.

Considering the complexity of today's software systems, and the increase in their development costs, errors detected after system delivery are very expensive to fix. Requirements errors are particularly expensive to fix at this late stage. About 50% of errors or changes required in a delivered software systems and 75% of the total cost of error removal are due to requirements [5]. According to the above reference, traditional software development models (waterfall life-cycle approach) lack the guarantee that the resulting product meets the real customer needs.

Requirements analysis and validation represent a bottleneck in the process of producing software systems that satisfy the customers' real needs. It is important to detect errors in requirements as early as possible. In terms of cost effectiveness, the cost of fixing an error during system implementation can be several orders of magnitude higher than fixing the error at the initial requirements analysis [82]. The problem of requirements analysis and validation is even more severe in large embedded systems. If these systems have strict real-time constraints, as is often the case, the problem is amplified.

The contributions of this dissertation addresses the problems identified above by providing the formalism and automated support. Both can assist in analyzing, validating, and controlling the evolution of system requirements with customers involvement. The result are requirements that satisfy customer real needs and a system design that consistently maps these requirements.

A. PROBLEM DEFINITION

The prototyping model is a recognized alternative to reduce the risk of a delivered product that does not satisfy user requirements [54]. Prototyping has the advantage of the customer involvement in the evaluation and validation process. Observing an executable model of the proposed system stimulates users to judge the degree the demonstrated system behavior meets their requirements.

Prototyping becomes even more powerful if supported by the appropriate tools to assist or automate parts of the process. The Computer-Aided Prototyping System (CAPS) is a set of tools for this purpose. CAPS is a software design environment that automates many activities in the prototyping process. It assists designers to quickly draft the proposed system design, generate and augment code, compile, and demonstrate this first cut prototype to customers. This is done using a graphical user interface and supported by a design database that contains the project information. It is also supported by a software base that includes reusable components and search and retrieval mechanism.

The convergence of the prototype behavior to the customer requirements is achieved through an iterative change/demonstrate/validate process. This process is greatly improved if combined with a formalism and automated support for the early part of the process. Specifically, the activities that start with the prototype demonstration and end by responding to the user responses to evolve to the next version of the prototype lack automatic support for many planning, control, coordination, and analysis activities that assist the customers, designers, analysts, and the managers. The goal of the work reported here is to provide such support. The intended support for responding to requirements changes should be provided within the context of a formal model rich enough to span not only the requirements analysis and evolution process but the whole system life cycle. The main outlines of the additional formalism and support to achieve this goal includes:

1. The formal representation of requirements and their evolution.
2. Linking the change in requirements to the system design to automatically expose the affected parts of this design.
3. Recording and reasoning with the design rationale.
4. Formalizing the demonstration process by providing the formal representation for and recording the participating stakeholders, test scenarios, and the stakeholders' criticisms to the demonstrated behavior of the prototype.
5. Analyzing the stakeholders' criticisms and synthesizing a set of issues based on these criticisms to be resolved. Resolution of these issues assists in the convergence of the design to the customers' real needs.
6. Automated assistance in identifying alternatives available to resolve open issues.
7. Assisting stakeholders in their evaluating the available alternatives to choose among them based on specified criteria.
8. Providing a formal mechanism that supports stakeholders in the deliberation process that precedes the final decision as to which alternative to choose. This same mechanism is used to quantify and formalize judgement and provide the representation for the group final decisions.
9. Providing other support functionality including inferring additional information from that given, specifying and enforcing constraints, and establishing relationships.
10. The automatic generation of proposed plans to modify the design as a consequence of requirements changes.

11. A design database capable of representing and encompassing this huge amount of the process knowledge in a way close to the conceptual data model.

These are the main elements and characteristics of the problem on hand. As a constraint on any resulting solution, the conceptual data model must build on, use, and be consistent with the related efforts done within the projects. Among these is the evolution control system (ECS) that represents a core in the project control functionality of CAPS.

B. CONTRIBUTION

The main contributions of this dissertation are:

1. The enhancement and extension of an evolution graph model into a conceptual model that incorporates the modeling of the requirements analysis and evolution process. This enables the extension of the project control functionality in CAPS to include requirements evolution and the automated link of this evolution to the design and implementation levels of the system prototype. Hence changes in requirements automatically expose the other affected parts of the system. Automatically generated plans are proposed to propagate the requirements changes into the other system parts. This also enables the recording of and reasoning with design rationales that span the whole system life cycle.

2. A new decision support mechanism based on the conceptual model and inference rules to support inferring additional information from that given, specifying and enforcing constraints, and establishing relationships. This mechanism also provides the support to the individuals involved in the software design process including
    a. Stakeholders from the customer side.
    b. Software project managers.
    c. Software analysts.
    d. Software designers.

3. Adapting and improving a model for decision making to support the judgement of stakeholders on choosing among alternative requirements changes. This model is

also used to combine individual judgements into a final group decision. This decision is then formalized into a change request to evolve requirements and expose the affected parts of the system design. This contribution enables the involvement of the customers directly in the process and formalizes their judgement on alternative requirements changes.

4. Extending and improving the Issue-Based Information Systems (IBIS) model for deliberation and design rationale capture on software requirements analysis. The improvements include:
   a. Increasing the formality degree of the model.
   b. Combining it with the model in 3. to quantify and formalize judgements.These improvements make the deliberation process more formal and based on quantified measures. It also enables the capture of more relevant information related to requirements change options, criteria of judgement on these options, and the outcome of the deliberation process. This fills gaps in the IBIS model that restricts its use effectively in our context.

5. The conceptual and architectural design of a project database capable of representing the model entities naturally along with time varying relationships that link these entities. Object-Oriented Database paradigm is used in this design. The design allows feasibility of practically implementing the proposed decision support facilities.

6. A new implementation model which provides more effective communication between Ada and C++ languages. This new technique enables Ada to use the facilities of the Object-Oriented database (ONTOS) despite the fact that ONTOS has no Ada binding. This step removes the main difficulties in implementing the design in 5. above. This is significant because ONTOS is one of the most advanced commercial Object Oriented (O-O) databases available and because none of the available O-O databases at this time supports Ada.

C. WHY DECISION SUPPORT IS NEEDED FOR REQUIREMENTS

The process of acquiring, analyzing, and evolving requirements is very complex. This complexity increases in our context. This is due to the tasks of determining the effect of the evolution on the other system parts as we explain in Chapter III. Therefore automatic support is needed. Some of the reasons behind this need are as follows:

1. The underlying structure of the problem at hand is complex and is characterized by time-varying relationships.
2. A huge amount of information is captured during the process that requires tracking beyond the capabilities of unassisted human individuals.
3. The automatic generation of analysis and design activities based on the existing or inferred dependencies requires automatic support.
4. What-if kind of analysis is inherent in the process to reach consensus and study different promising options.
5. Alternative generation and evaluation for impact is normally encountered.
6. Managing the project resources is a critical task for better utilization.
7. Scheduling activities related to the process requires automatic support.
8. Automated decision support is also needed for:
    - Automatic assistance for monitoring state transitions of activities.
    - Default action triggering and assignment of default values
    - Configuration management and evolution of the system (prototype) under development.
    - Controlling the dynamics for the ongoing changes during the course of design and development.
    - Computing or inferring values and relationships which are not explicitly stored in the project database.

D. ORGANIZATION OF DISSERTATION

The rest of this dissertation is organized as follows: Chapter II provides preliminary technical background and overviews related research works. In Chapter III we develop a formal model for requirements analysis and evolution. Chapter IV describes a decision support mechanism based on the conceptual model developed in Chapter III. This decision support mechanism is extended in Chapter V to provide a formalism that assists and quantifies stakeholders' judgements, and formalizes the decision making process. The conceptual design of a project database is described in detail in Chapter VI. Chapter VII provides a new implementation model that allows Ada use an object-oriented database (ONTOS DB) facilities despite the fact that this database has no Ada binding. Chapter VIII provides a detailed case study. Concluding remarks and directions for future study are given in Chapter IX.

II. TECHNICAL BACKGROUND AND PREVIOUS RESEARCH

A. ACQUIRING AND ELICITING OF SOFTWARE REQUIREMENTS

Requirements can be defined as a set of system capabilities that must satisfy a set of customer goals within a set of restricting constraints. The formulation of requirements should also take into consideration the operational environment under which these capabilities is provided.

Requirements analysis is the first phase in the software development process. It is a highly critical step in the software life cycle because of the inherent problems associated with the process. Incompleteness, contradiction, and ambiguities are examples of such problems [2]. Inconsistency is also a major concern in requirements analysis [24]. Unless these kinds of problems are identified and resolved in the requirements analysis phase of the software process, they can have very bad effects on the subsequent development steps and will be very costly to fix in later stages. For this reason requirements analysis should be done with great care and precision.

The requirements analysis process starts from an initial problem statement provided by the customer and proceeds in three stages, requirements acquisition, functional specification, and validation [2], [82]. Those three stages are not necessarily performed sequentially (see Figure 2.1). At each stage the knowledge explored may be used to feedback to the other stages. The initial problem statement is characterized as being informal, incomplete, and vague.

Requirements acquisition is perhaps the most crucial part of the software process [31] in part because it relies more on knowledge about the application domain about which the analyst has a limited knowledge, and the customer is not normally qualified to specify it accurately. Acquiring requirements starts with an elicitation process. Requirements elicitation is the process of identifying needs and bridging the disparities among the involved communities for the purpose of defining and distilling requirements to meet the constraints of these communities [22]. The primary goal of requirements elicitation is achieving a consensus among a group of customers about *what* they want. In the mean time the analyst tries to work out some simplified model of the proposed system that captures the concepts needed for describing the mini world in which the system will operate. Within the context of the requirements acquisition process the system goals are identified and elaborated along with the different kinds of constraints imposed on the system such as resource, performance, and implementation constraints.

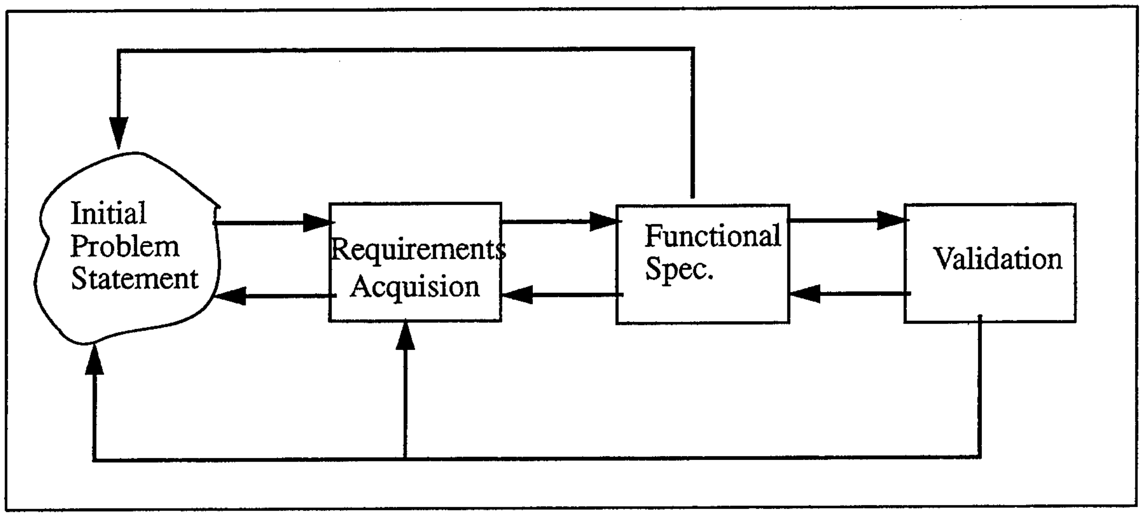


Figure 2.1 Requirements Analysis Phases

The goal of the functional specification stage is to construct a black-box model of the proposed system [82]. This model captures just the aspects of the proposed system behavior relevant to the users of that system. The output of the functional specification activity is a set of external system interfaces to the proposed system. During this stage of requirements analysis the effort is concentrated on answering the question *what* to build not *how* it is built.

Validating requirements is the process through which the customers' real needs are checked against the formalized requirements to make sure that the formalized requirements accurately meet those needs. In our judgement, this process should be continuous and spread during all activities of requirements analysis. There are three other separate steps involved in validating requirements [73]:

1. The requirements should be shown to be consistent; any one requirement should not conflict with any other.
2. The requirements should be shown to be complete including all functional and non functional requirements.
3. The requirements should be shown to be realistic; there is no point in specifying requirements which are unrealizable.

B. REVIW OF RELATED WORK

Hsia in [63] has identified nine research areas that have a significant payoff in requirements engineering and the relative time-frame during which work in these research areas is expected to have major effect on practice ranging from short, mid, and long-term. Two areas were classified as short term: improving natural language specification and prototyping. In the first area for example, work has been done to identify the attributes that a requirements writing team should look for when they review a natural language specification. Some pioneering work in this area is augmenting natural and formal language specifications with scenarios which are based on a formal model, and are generated, analyzed, and validated in a systematic manner [64]. Although the associated process provides for understanding, analyzing, and describing system behavior in terms of ways the system is expected to be used, this method currently can not deal with concurrent events, timing constraints, and interaction among scenario views. A scenario of system interactions with its environment is used by many of the object-oriented analysis methods [43] and it has been identified as one of the means for achieving separation of concerns in describing system behavior [55].

The second research area, prototyping (to be discussed in detail in the next section), has proved very effective in the requirements engineering process and is recognized as a part of the requirements process according to the IEEE standards [23].

Another research area in requirements engineering related to our work that was not reported in [63] is the work done on the deliberation process with the intent of capturing the process knowledge and using it for conflict resolution between different view points. The application of such approaches to requirements engineering serves in providing support for the elicitation process which can be viewed as a deliberation process between the stakeholders, specially early in the exploration of the customer needs. Most of these approaches employ ideas based on or similar to the IBIS model [35] for recording the argumentation related to the deliberation process. Extensions to IBIS model provided by these methods range from just augmenting the model with a hypertext-based tool to real extension of the model types and relationships as explained below.

1. IBIS Model

The Issue-Based Information Systems (IBIS) model was developed by H. Rittel [35] and is based on the principle that the design process for complex systems is fundamentally a conversation among the stakeholders (e.g., designers, customers, implementers) to resolve the design issues. According to this model, an ***issue** is any problem, concern, or question that may require a discussion for the design to proceed.* Each issue can have many positions where a ***position** is a statement or assertion that resolves an issue*. Each position is supported by or objected to by one or more ***arguments**.*

The deliberation process in the IBIS model is represented by a network where the nodes model issues, positions, and arguments, and the relationships among these elements are modeled by the links in the network (see Figure 2.2).

A typical IBIS process starts by one user posting an issue node and may also post a position node proposing one way to resolve the issue. He may support his position by posting an argument node too. Another user may post a challenging position supported by a set of arguments. Others may post other positions and/or arguments which support or object to any of the positions. Additionally, other issues may be generated during this kind of discussion and linked to the issues that suggest them. In the original IBIS, the intent of this deliberation is for a stake holder to understand other's view points or convince them of his.

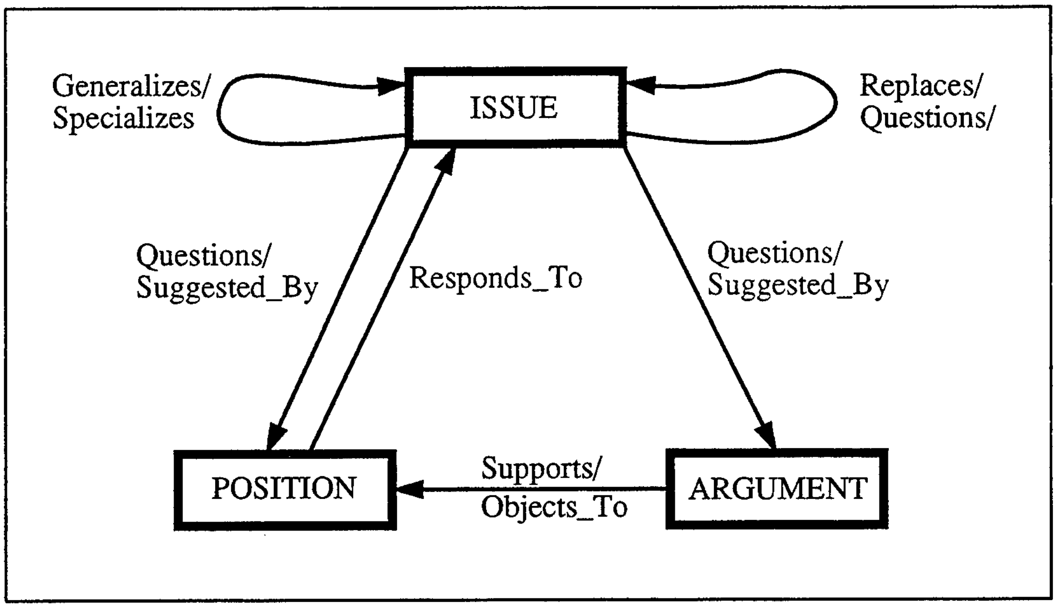


Figure 2.2 IBIS Model Types and Relationships

2. IBIS-Related Work

a. Inquiry-Cycle Model

This model provides the structure for describing and supporting discussion about system requirements. It is a hypertext-based model that captures the dynamically ongoing deliberation process. The model has three phases: Requirements Documentation, Requirements Discussion, and Requirements Evolution [15]. In the first phase, the proposed requirements are written by the stakeholders; each is a separate node in the hypertext. In the second phase, the stakeholders conduct the discussion by posing questions, answers, and possibly reasons that justify answers. The question-answer deliberation process is driven by a scenario analysis technique that complements the model to acquire requirements. The ultimate result of a requirements discussion is a commitment to either freeze a requirement or change it, and a change request may be generated accordingly. Notice that question-answer-reason is similar to the IBIS triple issue-position-argument.

In this model there is the underlying assumption that there is a requirements document available and the deliberation process is done within the context of a prepared set of scenarios that question the behavior of the proposed system. However scenarios have narrow coverage and the problem is multiplied if no requirements document is already available. According to [44], the model is refined with the aim of better support for collaboration through shared hypermedia. A group tool (EColabor) based on the refined model was developed which uses Internet and World Wide Web technology for its implementation. With this tool, the new instance of the Inquiry-Cycle model can handle multimedia information, represent requirements analysis more flexibly, and introduce a "reminder" type of discussion element that captures general and new ideas.

b. gIBIS

gIBIS (for graphical IBIS) is a direct implementation of IBIS model that also provides a hypertext interface to this model. It is designed to facilitate the capture of the early design deliberations. This hypertext system makes use of color and a relational database for building and browsing typed IBIS networks. Further, gIBIS supports the collaborative construction of these networks by the cooperating team members that may be spread across a local area network [35].

The gIBIS interface is divided into four tiled windows: a graphical browser, a structured index into the nodes, a control panel, and an inspection window. The browser provides a visual presentation of the IBIS graph structure as well as the ability to create new instances of the IBIS types in a context sensitive manner. The node index window provides an ordered hierarchical view of the nodes in the current IBIS network. Nodes can also be selected through the index as well as the browser. The control panel is composed of a set of buttons which may be associated with menus to extend the functionality of the tool beyond the simple node and link creation.

As the underlying representation of the knowledge in gIBIS is not formal enough, its ability to reason with that knowledge is severely restricted [9]. Further, gIBIS also inherits the IBIS problems of lacking the explicit representation of goals and the outcomes of the argumentation.

gIBIS is extended in [61] to superimpose issue-based and truth-maintenance [36], [40] techniques to provide a merged capability for recording design rationale. The intent of this superimposition was to combine the power of truth-maintenance approach for use with automated inference techniques and the power of the issue-based approach to express much of the informal and rhetorical information of the process. The new system presents the user with issue-based structures that can be annotated with informal information, and it provides an automated inference capability on these structures through an underlying truth-maintenance and expert system. The latter system allows the user to carry out what-if analysis by choosing different resolutions to design issues and it graphically shows the propagation of the belief status among the components of an issue-based system's style of display.

c. CoNeX

Coordination and Negotiation support for eXperts in design application (CoNeX) has been developed as an extension to the DAID's (Development Assistance for Intensive Database Applications) knowledge-based software information system ConceptBase project [81]. It provides a group collaboration facilities and integrate different tasks encountered in software projects. CoNex emphasizes integrating the semantics of the software development domain with aspects of group work, and the social strategies to negotiate problems by argumentation as well as assigning responsibilities for task fulfillment by way of contracting [81].

CoNeX is based on the integration of three conceptual models: a group model for task cooperation, a multiagent conversation model for task-oriented negotiation, and a software process data model. The first model handles managing teams of experts (analysts, designers, implementers, etc.) to execute a set of actions forming a plan. The second model is concerned with controlling the interactions that task-oriented groups often use to achieve or modify agreements. This is modelled by employing two techniques of conversation:

1. *Conversation for actions:* Messages are passed in order to assign plans to people, to make binding commitments, to implement a plan in terms of activities, and guarantee proper termination and acknowledgment of the task-oriented activities.
2. *Conversation for negotiation:* In this communication mode, opinions are exchanged in terms of debate in order to coordinate goals, and agree upon some plan or activity to be done through argument exchange and final decision making. This technique is the one based on IBIS model of argumentation.

The software process data model (called CAD^0 for Conversation among Agents on Decisions over Objects [76]) is introduced by the content-oriented part of the whole model. According to this model, the software process is viewed as a set of inter-related design decisions realized by agents through actions. The result of a decision is the transformation of input objects into output objects. It is also concerned with recording of administrative aspects of software objects, recording of semantic aspects of software objects including the semantic dependencies, and integrity control and partial automation of administrative and content-oriented actions. This model also employs the abstraction principles and deduction mechanism of the knowledge representation language Telos [42].

d. REMAP

The REpresentation and MAintenance of Process knowledge (REMAP) is a system that captures the history about design decision in a structured manner [5]-[9]. The underlying model of REMAP is an extension of the IBIS model to include more types and relationships. A type for requirements was added to the model to represent the goal of the deliberation process. Decision resolves an issue by selecting one of the latter's positions. Arguments are explicitly qualified by assumptions. Constraints generated or implied from the resolution process represent the linkage between the deliberation process and the creation of artifacts.

REMAP uses Telos as a conceptual modeling language which provides the capabilities to support the representation and reasoning with the process knowledge. Telos provides automatic inferencing to enable access to the knowledge implicit in the model and provide mechanisms to maintain the integrity of the knowledge base made up of interconnected components that are incrementally modified. Additionally Telos provides aggregation, classification and generalization mechanisms which are important features in any object-oriented representation of knowledge bases.

e. OSC

OSC is a design tracking tool based on an extensible conceptual model for recording design decisions and other supporting information. It consists of a design database and a family of query, manipulation, and extension facilities [25]. The design database maintains a record of the design decisions. With the extension facility, the design database schema can be extended and new query or manipulation facilities can be added. The architecture of this tool consists of three layers: a core that represents a seed version of the database schema, an environment-specific envelope that can extend or customize the core model, and an interaction envelope for customizing the user interface.

In the design database of OSC, the process knowledge is represented in objected oriented fashion. Object types include problem elements, design decisions, assertions, and agenda items. These types comprise the types of the seed version of the database schema that can be specialized according to the context. Problem elements are the same as the IBIS concept issue. Design decision represent information about possible actions and choices. Assertion objects capture the justification for the decisions made. Agenda items are reminder of such pending tasks as decisions that must be justified or assertions that must be verified.

f. Others

Other models that employ ideas based on or similar to the IBIS model are the IBE and SYNVIEW [9]. According to the aforementioned reference, these tools either lack the explicit representation of the context in which the argumentation occurs, or do not provide any reasoning to use the captured knowledge. IBE is a hypertext system based directly on the IBIS model with functionality similar to those of gIBIS with the addition of a document editor augmenting the hypertext. SYNVIEW [20] is a tool used to cooperatively support indexing, evaluating, and synthesizing information through interactions by many users with a common database. In the underlying model, an argument is represented by a ranked list of items of evidence for or against a particular conclusion.

g. Relation to our Work

IBIS-related models as applied to requirements engineering concentrate on the deliberation process without explicitly linking and using the process knowledge into the lower level design artifacts. These models address the acquisition phase of requirements engineering. In order to be practically useful, the mechanisms of these models should augment or be linked to other models related to change impact analysis, evolution control, decision analysis, etc. More expressive types and relationships make these models more capable of capturing more relevant information.

Our model provides more enhanced capabilities than those provided by the above models. We support not only the deliberation process as the case in the majority of these models, but also use the outcome of the deliberation to evolve requirements and expose the effect of the process on the other system artifacts. A plan for these system-wide changes is automatically generated in a proposed state for further consideration by the managers or analysts. In gIBIS, by contrast, the process ends by building the IBIS network.

Most of the above models are more suitable for recording the rationale behind decisions. For example, in the *OSC* tool, the underlying model is simple and can be used efficiently for tracking decisions. However, decisions are neither linked to the source of the problem for which the decision is made, nor to where they impact. Even with the extension facility, the process remains within that frame. We provide the support for recording design rationale as part of our model. In our case design rationale carry more information and is tightly linked to the design itself. Additionally, in our case problems (issues) are linked to individuals who raise them through criticism objects from one end and the rest of the system from the other end.

In another example, although REMAP extends the IBIS model and makes it more formal, it is classified with the models that deal with the up stream part of the process. It is mainly concerned with the deliberation process. There is also the implicit assumption that issues are given or generated during the deliberation process. In our work issues are synthesized and formalized from the responses of the stakeholders stimulated by the system demonstration. For each issue we provide the support for generating alternatives to resolve the issue. These alternatives are generated after analyzing the requirements affected by the issue. Stakeholders are also supported to select from these alternatives based on a formal technique. This is done within the frame of a formal model that explicitly provides the representation for the whole span of the analysis and design process not only the deliberation process as is the case in REMAP.

Other important issues we address in our work which are missing in the above models are the following:

1. Explicit representation of design artifacts at all levels.
2. Support for decision impact analysis in terms of revealing the consequences of making decisions.
3. Evaluation of the available options in more formal way.
4. Automated support for generating activities to execute the decisions.
5. Automated support for generating proposed plans that include the activities in 4.

As a final note, many of the IBIS-based models that address requirements are concerned with a static record of decisions made and their rationales that only serves for documentation purposes. What is needed, though, is an active model of the requirement decisions, the design decisions, and trade-offs and their evolution [62]. In our approach we address these issues within the framework of our model.

3. Non-IBIS Models

a. RA

The Requirements Apprentice (RA) is an intelligent assistant for software requirements acquisition and analysis [17]. The RA is developed as part of the Programmer's Apprentice project [16] whose overall goal is the creation of an intelligent assistant for all aspects of software development. The focus of RA is on the transition between informal and formal specifications [31] supporting the earlier phases of creating requirements in which ambiguity, contradiction, and incompleteness are inevitable.

Internally, the RA consists of three parts: Cake which is a system that provides the basic knowledge representation and automated reasoning facilities, the executive that contains algorithms and data structures specific to the RA, and the cliches library that contains reusable fragments of requirements and associated domain models represented as frame hierarchy. The executive handles interaction with the analyst and provides high-level control of reasoning performed by Cake. The RA is an assistant to requirements analysts and is not intended for use by end users. The analyst communicates with the executive by issuing commands. Each command provides fragmentary information about an aspect of the requirements being specified. The immediate implications of a command are processed by Cake, added to the requirements knowledge base, and checked for consistency. If the analyst makes a change in the description of the requirements, the executive incrementally incorporates this modification into the knowledge base, retracting the invalidated deductions, and replacing them with new deductions.

The interface to the RA displays three kinds of information [31]: the first window displays information about requirements as it evolves, a dialog window for entering commands by the analysts to the RA and displaying the latter immediate responses, and a third window that displays a list of pending issues that need to be resolved. The first window is basically a window into the knowledge base that can display information in many formats, and can be used to inspect the contents of the cliches library or inspect the reasoning behind the conclusions drawn by the RA.

In relation to our work, the range of users supported by the RA is limited to only analysts. Ours supports all stakeholders: customers, analysts, project managers, and designers. Also the feedback of the RA is to the analyst. This means the final requirements mainly reflects his point of view. In our approach the feedback is always presented to the customer through the demonstration. This makes requirements incrementally converge to the customers' real needs.

b. KBRA

The Knowledge-Based Requirements Assistant (KBRA) is a component of the knowledge-based software assistant (KBSA) [1], [21] funded by Rome Air Development Center. Within the KBRA environment, requirements are entered into the system in any order or level of detail in many different formats. The KBRA is then responsible for doing any book keeping to allow the user to manipulate the requirements while it maintains consistency among requirements.

According to [21], the KRBA capabilities include: support for multiple viewing options (e.g., data flow, control flow, state transition, and functional flow diagrams), management and editing tools to organize requirements, and the support for constraints and non-functional requirements through the use of spread sheets and natural language notations. KBRA can also perform requirements analysis to identify inconsistency and incompleteness as well as generating explanations and descriptions of the evolving system. To build and maintain a consistent representation of requirements, KBRA provides truth-maintenance support including default reasoning and dependency tracing.

According to [1], there is no strong guarantee that the final requirements meet the customers' real needs. This is due to the fact that KBRA does not have strong focus in customer validation. It was developed to only support requirements engineers [86] to enable them express requirements in a variety of ways suitable for the problem domain. The advantage our approach has is the involvement of the customers in the process from its beginning.

c. KAOS

In the context of KAOS (Knowledge Acquisition in autOmated Specification) project, the work on requirements focuses on a general approach for requirements acquisition driven by high-level concepts such as goals to be achieved, agents to be assigned, alternatives to be negotiated, etc. The underlying structure of the approach taken has three main components: a conceptual model, a set of acquisition strategies, and an acquisition assistant [2].

The conceptual model further involves three levels of modelling:

1. The meta level: refers to domain-independent abstractions and is used to model:
   - Meta concepts such as Agent, Action, Entity, Relationship, etc.
   - Meta-relationships that link meta-concepts e.g., Performs, Input, IsA, etc.
   - Meta-attributes of meta-concepts and meta-relationships e.g., Load of Agent, PostCondition of Action, etc.
   - Meta-constraints on meta-concepts and meta-relationships
2. The domain level: refers to concepts specific to the application domain and is made of concepts that are instance of the meta-level abstractions. For example in a library system as a subdomain of resource management application domain, a borrower is an instance of the Agent meta concept and CheckOut is an instance of the Action meta concept. These concepts are linked also by links which are instances of the meta relationships, e.g., Borrower performs CheckOut. Domain-level concepts must also satisfy instances of the meta constraints.
3. The instance level: refers to specific instances of domain-level concepts

The meta model is represented as a graph where each node represents one of the meta types (e.g., goal, action, agent, event, entity), and where the edges capture the semantic links between such types. According to this modeling, the requirements acquision process corresponds to some way of traversing such a graph to acquire appropriate instances of the various nodes and links according to the underlying constraints.

Acquision process is governed by strategies telling which way to follow systematically in the graph. An acquisition strategy is a composition of steps for acquiring components of the requirements model as instances of the meta model components. For example, the graph can be traversed backward from the goals to be satisfied by the system, backward from the agents available in the system and their respective views, or backward from a supplied set of scenarios. The strategy considered in [2] is the first one; goal-directed.

The automated assistance provided is built around two repositories: a requirements database, and a requirements knowledge base. The first maintains the requirements model built gradually during the acquisition and can be queried normally. The second one contains domain level knowledge and meta-level knowledge. The domain level knowledge is organized into specialization hierarchy where requirement fragments for a specific class of application can be inherited from more general applications and from more general tasks. The meta-level knowledge concerns more general aspects like ways of conducting specific acquisition strategies including tactics that can be used within strategies e.g., "prefer those alternatives which split responsibilities among fewer agents".

This model is very rich and its meta part can capture requirements knowledge for wide range of applications. One of the few drawbacks of this model is the complexity of the underlying structure. This makes it difficult for the model and implementations based on it to be used in effort reduction. Although the span of the process covered by our model is wider (the whole life cycle) than that covered by that model, our model is simpler.

4. Deductive Database Approach

We have explored the use of deductive database model as a formal model for representing, storing, and reasoning with requirements knowledge because the underlying structure was seemingly promising. The idea was rejected for reasons to be explained later in this subsection. A deductive database (DDB), known also as an expert database, or a logic database, is a database that is managed by a deductive database management system [12] (DDBMS) that supports the proof-theoretic view of the database. By applying the deductive rules that comprise a part of the database intension (schema), additional information can be deduced from the extension of the database. The main difference then between non-deductive database and a deductive one is that in the first, querying a database, we obtain facts that have been directly stored or combination of them. In a DDB it is possible to obtain not only the facts directly stored, but also new facts using inference rules along with other domain rules stored in the database. To preserve a correct state of the database (deductive or non), we also include integrity constraints with the database that serve to maintain the states of the database within a permissible set.

A proof-theoretic view of a database (opposed to model-theoretic view) [30] is informally obtained by constructing a theory *T* that admits the extension of the database as a unique model [14]. The construction of *T* is also based on the following set of assumptions that govern query and integrity constraint evaluation of the database, deal with the negative representation of facts, and make the universe to which queries refer more precise:

1. The closed world assumption (*CWA*): facts that are not known to be true are assumed to be false.
2. The Unique name assumption: individuals with different names are different.
3. The domain closure assumption: there are no other individuals other than those on the database.

A (definite) DDB is then defined as a particular first-order theory *T* along with a set of integrity constraints. A definite DDB allows only function-free and Horn definite clauses while an indefinite DDB allows only function-free non-Horn clauses including negative clauses [45]. The CWA leads to inconsistency when used with indefinite DDB. For this reason Minker [41] introduced a generalization of the CWA (GCWA) to deal with negative information in the indefinite DDB. The theory *T* consists of the following proper axioms:

1. The unique name axioms.
2. The domain closure axioms.

3. Equality axioms which specify the usual properties of equality: reflexivity, symmetry, and transitivity, and principle of substitution of equal terms. These axioms are needed because other axioms use them.
4. The completion axioms that effectively represent the CWA.
5. The elementary facts axioms which is a set of ground atomic formulas each corresponds to a tuple in a relation DB table.
6. The deductive laws axioms which is a set of function-free definite clauses. Definite here means that these rules do not include implications with disjunctive conclusions because they create indeterminacy. As an example of an indefinite rule, consider the implication of the form $a \Rightarrow b \vee c$, even if we know that a is true, we can ascertain neither the truth of b nor the truth of c independently.

Under some assumptions and with the intent of obtaining an operational database, some of the above axioms can be excluded or substituted by metarules, refer to [30] for details. According to this scheme the evaluation of queries and satisfiability of integrity constraints remains intuitively similar to conventional database schemes. A query in a DDB is a first order formula $(W \Rightarrow \ )$ where W is termed the body of the query. Any free variable in W is assumed to be universally quantified at the front of the query. An answer to the query $W(x_1,\ldots, x_p) \Rightarrow$ where the x_i are free variables, is the set of tuples $(c_{i1},\ldots, c_{ip})$ such that these set of tuples are derivable from the theory T.

An integrity constraint is a closed first order formula. A DDB obeys the integrity constraints if and only if every formula of the integrity constraints set of axioms are derivable from T, i.e., the integrity constraint is a logical consequence of the database. Integrity constraints can be checked after database updates by running them as queries. Thus if W is an integrity constraint, the database satisfies (violates) W if running $W =>$ as a query succeeds (fails). The deduction capability of the DDB comes from the axioms set of the deductive rules where tuples are not only those explicitly stored in the database, but also those that can be derived using the deductive rules.

The DDB modeling relies on the first order logic as a knowledge representation scheme where facts and relationships between facts are represented as logical formulas in the database. The benefits of logic as a knowledge representation scheme include [65]:

1. Logic is precise and unambiguous.
2. Representation uniformity; facts, implications, queries, etc. are all expressed in the same first-order language.
3. Operational uniformity where first-order proof theory is the sole mechanism for query evaluation and the satisfaction of integrity constraints.
4. Generality of inference and proof procedures.
5. Well-defined semantics.

Despite these advantages, the use of logic as a representation scheme has three main disadvantages [57]:

1. Although the representation capability of logic is powerful, it is limited (or the formalization process required is difficult) with respect to basic knowledge representation requirements. Standard first-order logic can not be easily used to represent such real world knowledge as beliefs, defaults, and incompleteness.
2. Procedural and heuristic knowledge is difficult to represent in logic. Procedural knowledge is critical for any integration of knowledge and data, particularly for knowledge acquisition and manipulation.
3. Logic lacks organizational and modularity principles which are crucial for large and complex applications.

When we started investigating the use of DDB modelling in the requirements analysis domain, we were motivated by the capabilities the approach can provide:

1. The DDB being a database in the first place can provide the capability of storing, retrieving and manipulating the requirements process knowledge which is classified

as data-intensive process.

2. Logic as the representation scheme in DDB provides unambiguous representation of knowledge which is a crucial point to deal with in requirements analysis.
3. Within the context of DDB, it is relatively easy to detect inconsistency [28] and contradictions that are difficult to do using other schemes.
4. Constraints can be easily modeled by considering them as special integrity constraints that have explicit representation in the DDB context.
5. Reasoning with the available knowledge to draw conclusions using inference capabilities [88] is directly available by using a DDB. This capabilities provide a mechanism to be used in providing the decision support for the requirements analysis process.

However, one main point discouraged us from using DDB modeling in the requirements analysis domain: only function-free formulas are allowed [66] in DDB as explained above. This limitation hinders the representation and manipulation of more general forms of knowledge instead of only constants and variables. Functions are excluded in DDB models to have finite and explicit answers to queries.

5. CAPS Graph Model

Software evolution in computer-aided prototyping is essential. This comes from the nature of the prototyping process [53]. Until agreed upon by stakeholders, a prototype is subject normally to frequent changes. Therefore computer support for evolution is important. The CAPS graph model is a data graph model for evolution that records dependencies and supports automatic project planning, scheduling, and configuration management[48]. According to this model, the evolution process of a software system is represented by a graph that at any given moment models the current and the past state of the software system. A typical instance of that graph consists of software objects that comprise the system configuration and the evolution activities (steps) applied to these objects.

The graph model views a software evolution process as a partially ordered set of steps. Each change in the system design from the moment it is proposed is performed within the context of one or more steps. Steps have states that reflect the dynamic progression of the change from the moment it is proposed until it is completed or abandoned (rejected). When rejected, the history of the activity remains in the project database. When completed, a step outputs new version or versions of the subject software component that underlies the change.

An earlier and primary version of this model was developed within the CAPS project [32], [33]. Luqi refined and elaborated the model in [47]. This same model was further enhanced, augmented with a scheduling model, team coordination mechanism, and implemented in [69]. We will return to this model for more detail in the next Chapter in the course of the development of the requirements analysis and evolution model.

C. THE ROLE OF PROTOTYPING IN THE SOFTWARE DESIGN PROCESS

Software process models refer to the activities involved in software development and maintenance termed as the software life cycle. The waterfall model is the most well known of these. Following are highlights of some of these models: [73], [74].

1. The waterfall model: This model views the software process as a series of consecutive phases such as requirement, specification, design, implementation, testing and maintenance phase.
2. Build-and-Fix model: In this approach a working system is rapidly developed, then repeatedly modified until it reaches an adequate functionality. This model is used where detailed requirements cannot be specified and where adequacy rather than the correctness is the main goal of system designers

3. Incremental model: in the incremental model, a system is designed, implemented, integrated, and tested as a series of incremental builds. A specific build consists of code pieces from various modules that interact together to provide a specific functionality.
4. Prototyping: This approach is similar to the build-and-fix model, but the main goal is establishing the system requirements. This normally followed by an implementation of the requirements to obtain a production quality system.
5. Formal transformation: In this approach a system development is treated as formal process that transforms the problem specification into the envisioned system in a discrete series of steps. Each step in this process corresponds to the application of a meaning-preserving transformation [83].
6. System assembly from reusable components: This approach uses the assumption that systems are mostly made up of already existing components. This means that the system development becomes an assembly rather than a creation process. However in our view the assembly process should be preceded by some other phases in the design process to determine what reusable components fill a required functionality. This approach when combined with rapid prototyping greatly enhances the software development process [19].

The Waterfall model of software development, is widely known and has been used with some success on a wide variety of products, however there have been also failures [74]. The Waterfall model as shown in Figure 2.3, introduced a phased approach that produces a series of documents containing requirements, specifications, and designs before detailed implementation of the system. The main problem with this approach is the assumption that system requirements can be discovered and frozen before implementation. This assumption has been found to be invalid in practice which may result in paying a very expensive cost in terms of time and budget to fix errors detected late during the test phase near the end of the project. This lack of any guarantee that the resulting system will meet the customer's needs is the main encouraging point for using prototyping model.

The risk inherently associated with the waterfall model can be greatly reduced by following the rapid prototyping paradigm. The purpose of software prototyping is to help customers understand and criticize proposed systems and to explore the new possibilities that computer solutions can bring to their problems in a timely and cost effective manner. Prototyping model will be elaborated more in the following section along with an operational project that employs this paradigm.

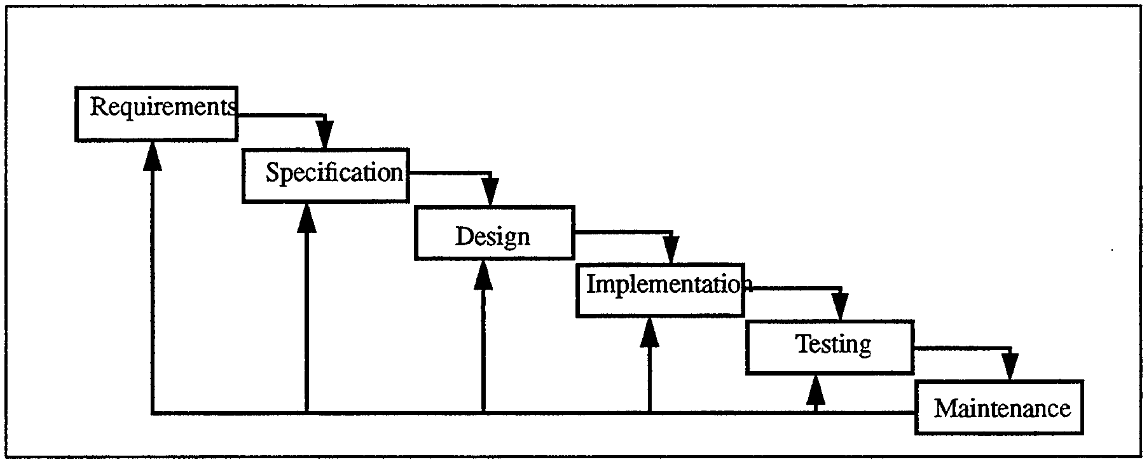


Figure 2.3 Waterfall Model

D. HOW PROTOTYPING CAN ENHANCE THE REQUIREMENTS ENGINEERING PROCESS

The prototyping model of software development is based on an iterative guess/check/modify cycle that relies on prototype demonstrations and customer reactions to the demonstrated behavior of the prototype. The main goal of this iterative process is to breach the gap between the customers' real needs and those same needs as understood by the designers and expressed by the behavior of the demonstrated prototype. An important outcome is a consensus about the requirements of the system to be developed before expending any further effort on the other development tasks [50].

The software development process based on the prototyping model is shown in Figure 2.4. There are two main phases that can be identified in the prototyping model [54]:

prototype evolution and production code generation. The main purpose of the first phase is to achieve a consensus on the requirements before investing any effort on implementation and optimization. The second phase may not exist if the prototype is of a throw-away type as we explain below. If it does exist, the main purpose of the code generation phase is to generate an efficient implementation when the requirements are stable. Even after product delivery, the prototype can be used to incorporate requirements changes into the working system by doing other iterations through prototype evolution and code generation phases.

The prototype evolution phase includes the shaded activities in Figure 2.4. The process starts with rapid analysis to determine an initial version of the requirements which is used to design the prototype system. The constructed prototype may represent only a subset of the requirements due to deliberate focus on critical aspects of a large system such as time constraints in a real-time system. The behavior of the prototype is then demonstrated to a group of the system stakeholders. The stakeholders may object because some aspects of the demonstrated functionality do not reflect their needs or because some required functionality has not been reflected at all. These user reactions are recorded and analyzed for cost implication and the goals of the project sponsors, and possibly triggering review and adjustment of these goals. A set of requirements changes is proposed based on the results of this analysis. The designers then modify the prototype to reflect the subset of the proposed requirements change that are approved by the manager of the prototype evolution effort. The modified behavior of the prototype is then demonstrated again repeating the same cycle until a consensus is reached on the customers' real needs.

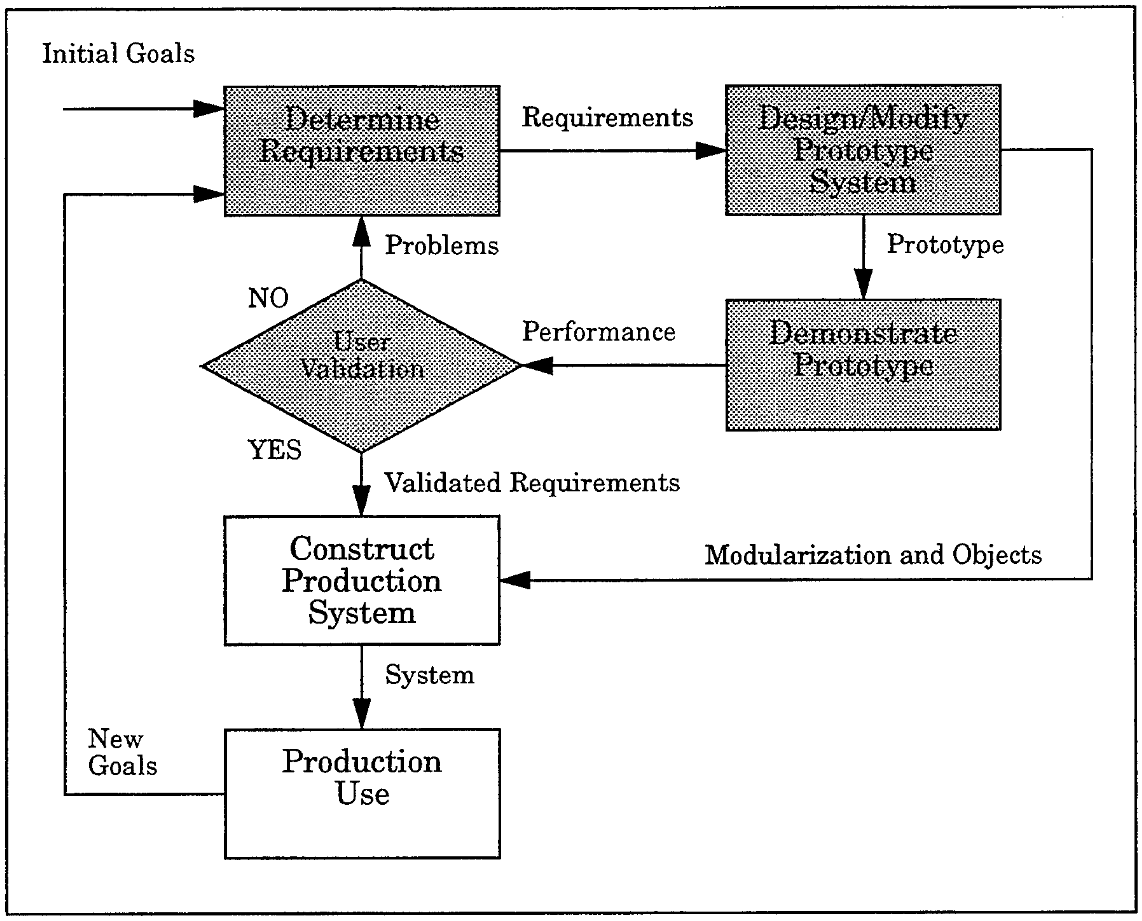


Figure 2.4 The Prototyping Process Model

The reader should have noticed that the prototype evolution phase of the prototyping process includes the three main tasks of requirements analysis: acquisition, functional specification, and validation.

1. Prototyping Approaches

Prototyping is currently practiced using one of two approaches; throwaway or evolutionary [54]:

a. Throwaway

The main purpose of a throwaway prototype is to be used as a tool for requirements analysis. After an agreement has been reached on what the customers' real needs are, the prototype is thrown away. In spite of the apparent disadvantage of wasting the effort, this approach may be useful in situations such as demonstrating the feasibility of new approaches or to convince a potential sponsor to fund a proposed development project. Other than that, the use of this approach is driven by the inadquacy of tools sophisticated enough to make use of the prototype by providing the required support.

b. Evolutionary

The availability of powerful supporting tools encourages practicing the evolutionary prototype approach. In this approach the prototype evolves through a series of versions. Each version except the first one is designed using the previous version(s) along with the feedback from analyzing the user reactions to the behavior of the demonstrated prototype. After a number of iterations, the prototype behavior converges to an acceptable behavior from the customer side. The number of iterations depends on many factors such as the complexity of the problem on hand, the skill of the designers and their knowledge about the problem domain, etc. Supported by the appropriate tools, the prototype or parts of it can then be incorporated into the code production process of the system development. Examples of such tools are tools that do the necessary adjustments and transformations to generate and optimize code aided by other tools to e.g., lookup and retrieve reusable code modules from a software database to fit some functionality of the proposed system.

E. OVERVIEW OF CAPS

The Computer-Aided Prototyping System (CAPS) [49] is a software engineering collection of tools for developing prototypes of real-time systems [52]. CAPS is an environment which automates the shaded boxes in Figure 2.4. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the Prototype System Description Language (PSDL) [51], which provides facilities for modeling timing and control constraints within a software system. CAPS is a development environment, implemented in the form of an integrated collection of tools, linked together by a user interface.

A CAPS prototype is initially built as a CAPS data flow diagram and a corresponding PSDL program. The CAPS data flow diagram and PSDL program are augmented with timing and control constraint information. This timing and control constraint information is used to model the functional and real-time aspects of the prototype. The CAPS environment provides all of the necessary tools for engineers to quickly develop, analyze and refine real-time software systems. The general structure of CAPS is shown in Figure 2.5.

The user interface provides access to all of the CAPS tools and facilitates communication between tools when necessary. The tools as shown in the figure are grouped into four sections, *Editors*, *Execution Support*, *Project Control* and *Software Base*. Details of the CAPS tools can be found in [49], in the following we give brief description for these tools.

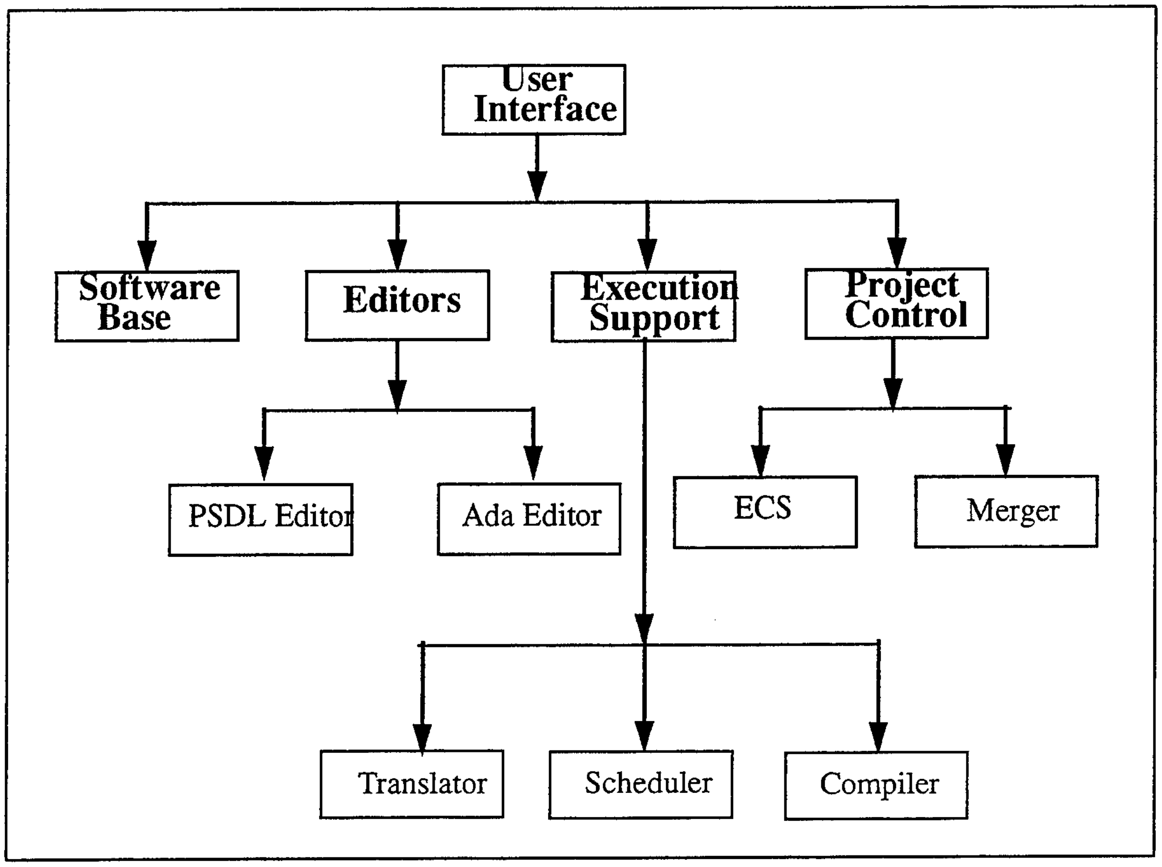


Figure 2.5 CAPS Development Environment

1. Editors

CAPS includes a number of different types of editors, the PSDL Editor is the heart of CAPS prototype design. This editor consists of 3 separate parts: the Syntax Directed Editor, the Graph Viewer, and the Graphic Editor. This tool allows the designer to create the CAPS data flow diagram and PSDL program, and assign all timing and control constraints to prototype components. CAPS also provides a selection of text editors facilities. Prototype designers can choose from vi, emacs and the Verdix Ada Syntax directed editors for editing Ada programs. The CAPS user interface provides a convenient file selection lists based on the currently open prototype.

2. Execution Support

The Execution Support group of utilities includes a translator, a scheduler and a compiler. The CAPS translator converts a PSDL program into compilable Ada packages which implement supervisory aspects of the prototype. The translator expects a complete PSDL program as an input, and creates several packages which make up, in part, the *supervisor module* of the prototype. The Ada implementation packages for the leaf components of the prototype components structure referred to as *atomic* operators are not generated by the translator. These must be either extracted from the software base, or custom-made by the designer.

The CAPS Scheduler is the tool responsible for determining the schedule feasibility for prototypes based on timing constraints assigned by the designer to the components of the prototype. Information is provided to the scheduler via timing constraints from the prototype's PSDL program. A prototype must be translated before it can be scheduled, and scheduled before it can be compiled. Upon scheduling a prototype, CAPS provides schedule diagnostic information which can be analyzed and used to direct timing constraint modifications.

The current version of CAPS uses the SunAda compiler. The compilation process is completely automated via the "Compile" command provided in the "Exec Support" pull-down menu in the CAPS User-Interface.

3. Project Control

Currently, the Project Control section of CAPS comprises two tools; the Evolution Control System (ECS) [69] and the Merger [18]. The ECS is a system that supports distributed prototype development in a team environment. The ECS makes use of a design database (DDB) for persistent storage of prototype development data. The ECS supports maintenance of a designer pool from which to draw for prototype development tasks.

Within the ECS, prototype development is modeled as a series of *steps*, which the project manager creates. These steps are automatically scheduled and assigned to available designers.

As part of our work, we extend the functionality and broaden the scope of the ECS to include the process of capturing the user reactions to the behavior of the demonstrated prototypes, synthesizing these reactions into issues to be resolved, and hence automatically generating activities to analyze the alternatives available and carry out requirements changes if necessary. The requirements changes automatically induce a chain of activities to propagate the changes down to the affected parts of the system design and implementation. These propagated changes are carried out by the ECS.

The CAPS Merger provides automated prototype *change-merging*. Based on *slicing* theory, applied to PSDL programs, the Merger automates the combination of two separate modifications to a base prototype. The Merger detects and warns of conflicts between the two changes to be merged. If no conflicts occur, or if they are overridden, the Merger creates a PSDL program for the newly created prototype which incorporates the changes of each of the modified prototypes.

4. The Software Base

The CAPS software base and its associated retrieval mechanism provide access to a repository of reusable Ada and PSDL components. The software base allows a designer to browse as well as query its components. Queries to the software base can be in the form of keywords or PSDL specifications. In the current release of CAPS, the software base matching mechanism is based on parameter matching.

F. OUR EXTENSIONS TO CAPS

Our work extends the capability of CAPS in multi dimensions. It extends the functionality and scope of the ECS to include the requirements evolution and analysis process within the context of a formal model. The current version of the ECS addresses only the design and implementation aspects of the system. With our extensions the early part of the prototyping process becomes more formal. This is the part mainly concerned with the activities of the user validation and the requirements determination driven by the prototype demonstration. See "The Prototyping Process Model" on page 33.

Our extensions to the CAPS graph model enables linking requirements to the system design and implementation to provide the automated support to expose the consequences of the requirements changes on the system design. The extensions and enhancements we develop support recording, analyzing, and resolving customers' concerns.

Based on our model and inference rules we also enables the augmentation of CAPS with a decision support mechanism capable of providing the automated support for establishing relationships, specifying and enforcing constraints, generating and evaluating alternative requirement changes, and maintaining consistency system wide.

We also provide automated support based on a design rationale capture model integrated with a decision making formalism. The design rationale capture model provides the automated support to:

1. Track the development history of the system prototype.
2. Support the design of new needs.
3. Maintain justification for design decisions.
4. Support the reuse of design artifacts.
5. Can be used as an aid to study designs.

The decision making formalism provides the automated support for stakeholders to evaluate and choose among alternative requirements changes. This support includes the formal quantification of the stakeholder judgements, and combining individual judgements into group decision. This same formalism can support analysts and designers to resolve design issues or analyze design alternatives. It can also support managers to set priorities.

Our work also provides CAPS with a conceptual and architectural design of a design database capable of:

1. Representing the prototype design entities and relationships naturally.
2. Providing efficient storage and management of the process knowledge.
3. Establishing feasibility of practically implementing the proposed decision support facilities.

An important enhancement to CAPS is the one enabled by our new implementation model. This new model:

1. Removes the main difficulties in implementing the conceptual design of the design database.
2. Provides a safe and efficient way to communicate Ada and C++.
3. Builds a partial Ada binding for ONTOS.
4. Reduces coding effort.
5. Simplifies code design.
6. Increases the productivity and uniformity of database applications development within CAPS.
7. Increases usability of code fragments.

III. A MODEL FOR REQUIREMENTS ENGINEERING VIA RAPID PROTOTYPING

A. THEORETICAL BASIS

Rapid prototyping is intended mainly to firm up user requirements in an interactive way between the prototype designers and the customers. Current models of this process are not formal and detailed enough to provide much decision support for the requirements analysis aspects of the process. Specifically, the activities that start with the prototype demonstration and end by responding to the user responses to evolve to the next version of the prototype lack automatic support for many planning, control, coordination, and analysis activities that help both the designers and the managers. The intended support for responding to requirements changes should be provided within the context of a formal model rich enough to support the following aids and capabilities with a reasonable degree of automation:

1. Planning the prototype demonstration.
2. Mapping user criticisms into the primitives of a formalized model to be analyzed and elaborated so that they can be synthesized into a set of issues to be resolved.
3. Analyzing alternatives available and choosing among them to make necessary modifications in the design to resolve the open issues.
4. Creating analysis activities as well as planning and executing these activities when the needed resources are available.
5. Controlling the evolution of the requirement components which are directly affected as well as propagating the implied effects of the changes and configuring the whole requirements hierarchy accordingly.
6. Propagating any changes in requirements to the affected parts of the system design and implementation.
7. Coordinating the effort of the design team.
8. Controlling versioning and configuration management to faithfully reflect the

intended effect of the dynamic ongoing changes.

A formal model that can encompass such a large range of decision support and provide the vehicle for representing the process knowledge can be partially provided by the graph model introduced in [47]. This model represents the evolution history, current state, and future plans for evolution as well as dependencies between the parts of the model. The graph model can cover multiple systems that share components, alternative variations of a single system, and a series of configurations representing the evolution history of each alternative variation of a system. This same graph model was augmented and enhanced in [69] to be used as a conceptual model for developing an Evolution Control System (ECS) that provides some of the support functionality. Currently the ECS supports activities 7 and 8 above. The ECS can also provide the automatic support for incorporating and propagating changes in software design components. However the automation provided by the current version of ECS does not directly address requirements or the upstream portion of the process. In the following subsection we summarize the original graph model and outline some enhancements to address the issues outlined above. We use these enhancements to provide additional support for requirements analysis in the context of rapid prototyping. The decision support functions enabled by these enhancements to the model is discussed in Chapter IV.

1. The Primitives for a Conceptual Model

The graph model [47] is composed of two main types of elements: software components and evolution steps, called components and steps below. Components are immutable copies of software objects that cannot be automatically generated on demand. Components can be of many different types. Both components and steps can be decomposed into hierarchies of like parts. Steps represent activities that comprise the analysis, planning, coordination, and design implementation of a request for a change. In our context a request of a change is derived from the justifiable user responses to the demonstrated behavior of the prototype. Evolution steps have the following properties [47]:

1. Every step is either a top level step or part of a top level step.
2. A top-level step represents the activities of initiation, analysis, design and implementation of one change request.
3. A step is *composite* if it has substeps and is *atomic* otherwise.
4. The inputs and outputs of a step are components.
5. An atomic step produces at most one new version of a system component in [47]. This restriction is relaxed in [69] to account for the cases where a designer who has been assigned an atomic step needs to decompose the assigned module, thus creating several new subcomponents.
6. The inputs and outputs of a composite step consist of the inputs and outputs of its substeps.
7. The model includes steps that have not been completed. Some of these are proposed as hypothetical alternatives, are in progress, or have been abandoned before completion.
8. Completely automatic transformations are not considered to be steps and are not represented in this version of the model.
9. A scope that identifies the set of systems and variations to be affected by the step is associated with each evolution step. The scope is used to determine which induced evolution steps are implied by the approval of a change request.

2. Input to Steps

Inputs to an evolution step are classified as either *primary* or *secondary* (*non-primary).*

a. Primary Input

The primary input to a step is the previous version of the component being updated by the step. An input to a step was originally defined to be primary if and only if it is the previous version of the same variation of the same object as the output of the step. Variations of an object represent parallel lines of development for the object that correspond to alternative design choices. The original definition restricts the output of a step to be in the same variation line as the step's input. In [69] this restriction was relaxed to allow an input to a step to be primary whether the output version is on the same variation as the input of the step or splits off a new variation. This modification makes some object versions belong to one or more variations to help trace the evolution history of each variation to the initial version of each object. We further relax the definition to allow more than one input to be primary to cover the case where several parallel lines of development are combined by merging operations.

b. Secondary Input

An input to a step is secondary if the designer needs read-only access to the component to accomplish the step. This kind of input is determined from the dependencies that link the primary input of the step and other model components.

3. Induced Steps

One of the key contributions of the model is to propagate changes in all parts of the system consistent with semantic dependencies on system parts affected by a change. When a step modifies a component, it induces other steps to carry out the changes in every other component affected by the original change. These induced steps are automatically generated based on dependencies between the being modified component and other components. The resulting induced steps are further analyzed by the manager and are adjusted to account for newly created and/or deleted structures.

4. Dependencies

Since both components and steps can be composite, the graph model provides primitives for representing the decomposition hierarchies of composite steps and components. Other kinds of dependencies among the nodes of the graph either of the same type (component-component or step-step) or different type (step-component or component-step) can be represented as edges of the graph, see section B.2 of this Chapter.

5. Step States:

Each step is in one of six states listed below. Transitions from one state to another corresponds to management decisions. Refer to [47] for the state transition diagram, its augmentation [69], [70], [71] and rules for determining implied transitions of substeps.

1. *Proposed*: The initial state of a newly created step. In this state a step is subjected to cost and benefit analysis.
2. *Approved*: In this state, the work to be accomplished by the step has been approved by the management and scheduling attributes such as priorities, required skills, and effort estimates are determined.
3. S*cheduled*: In this state, the step has been scheduled for implementation and expected starting and finish times are calculated.
4. *Assigned*: In this state a step is assigned to a designer and the work is in progress.
5. *Completed*: In this state the output of the step has been verified, and a frozen version has been entered into the project database.
6. *Abandoned*: This state represents a step cancelled before it has been completed. It is reachable from all other states except the "Completed" state.

We have kept our extensions to the original model minimal to preserve its simplicity. The enhancements have been necessary to capture information that is essential for carrying out the requirements analysis tasks we are investigating. As we describe in the following sections, the enhancements add new types of nodes, identify more software component classes different in their semantic contents, introduce another kind of step, and identify more relationships.

B. THE REQUIREMENTS EXTENSION OF THE GRAPH MODEL

Our model extends previous work to formalize and support the process that connects criticisms elicited by prototype demonstration to the changes in the prototype design. The main activities in this process include the tasks of planning and coordination of the prototype demonstration, recording and analyzing user responses to the demonstration and generating the activities to change the affected requirement components. These activities will automatically trigger a cascaded change according to the semantic or other dependencies that tie the system structure. Linking the criticisms into the model also serves to record and refine design rationale. Design rationale can be used in many ways, e.g., for redesign when perceived goals change. The rest of this section describes the extended model.

1. The Graph Nodes

The node types of the graph are refined as follows:

1. A new type of node represents the persons directly involved in the process either from the customer side or from the design team and management side.
2. Two kinds of steps are distinguished: analysis and design steps.
3. The class of software components is broadened to include more types which differ only in the semantics of their content attributes.

a. Analysis Vs. Design Steps

The new model distinguishes two kinds of steps: analysis and design steps. While both share the general properties and characteristics and are subject to planning, scheduling, control, and task assignment activities, the following semantic differences are identified:

1. *Analysis Steps*: address requirements and are controlled by the design team manager or analyst. Analysis steps are mainly concerned with:
   - Preparation for the prototype demonstration, which includes choosing concerns to emphasize and scenarios or test cases.
   - Analyzing and refining the set of criticisms posed by the customers in reaction to the behavior of the demonstrated prototype.
   - Establishing the link between the refined set of criticisms and issues they address. This activity may lead to the generation of new issues and may change existing ones. Issues are questions that must be resolved to determine the requirements.
   - Establish the links between the set of issues and the requirement components which are affected by the issues. This activity may lead to the generation of new requirement components and/or modify existing components
   - Provide information that assists in determining the alternatives (if any) available for resolving an issue.
2. Design Steps: address the design and implementation of the prototype, controlled by the manager and implemented by the designers. Previous versions of the model assume all steps are design steps. Design steps are mainly concerned with implementing the actual changes in the prototype design induced by a set of requirements changes.

This distinction was introduced primarily because management approval of an analysis step usually does not automatically imply approval of the implementation effort corresponding to the proposed requirement changes resulting from the analysis step, especially if more than one alternative is proposed as a result.

Additionally, all analysis steps associated with responses to the same demonstration must complete before any design step can start. This is to ensure that interactions among criticisms elicited by a demonstration have been determined before commitment to particular requirements changes for the next demonstration. Moreover, analysis steps are subject to a serialization constraints to guarantee that demonstration step completes before the issue analysis steps can start, and the issue analysis steps all complete before any requirements analysis step can start. This serialization is necessary because the decomposition of the next step depends on the output of the preceding step. See "The Process" on page 52.

A related issue is the choice between analyzing criticisms each within a separate substep induced from a top level step or analyzing all criticisms within the context of one analysis step. We prefer the second option where all criticisms are analyzed within one activity because analyzing each in isolation from the rest makes it difficult to check for contradictions and redundancies, and to properly account for interactions between requirements changes.

b. Software Components

Our model distinguishes among software component classes that represent customer criticisms to the prototype demonstration, issues to be resolved, requirement components, specification modules, and implementation modules. These components have the same attributes but differ in the semantic interpretation of their content attributes.

2. Relationships

The edges of the graph represent the different relationships that relate the model elements to each other. These relations as well as their inverses are important for the application of the model. Figure 3.1 depicts the relationships among different classes of components. We distinguish "user" as a specialization of the *Human* type because the analysis process is initiated by user criticisms to the demonstrated prototype and because users (unlike designers) are not assigned responsibilities for carrying out evolution steps.

Some other types of relationships will be introduced in Chapter V. These relationships mainly link the issue to be resolved with the stakeholders, their individual positions, justification arguments, alternatives available for deliberation, criteria for judgements, and the final group decision in the form of a change request. We discuss all these relationships in the course of extending and enhancing the IBIS model to give it more expressive power. The enhancement also quantify judgements using IBIS.

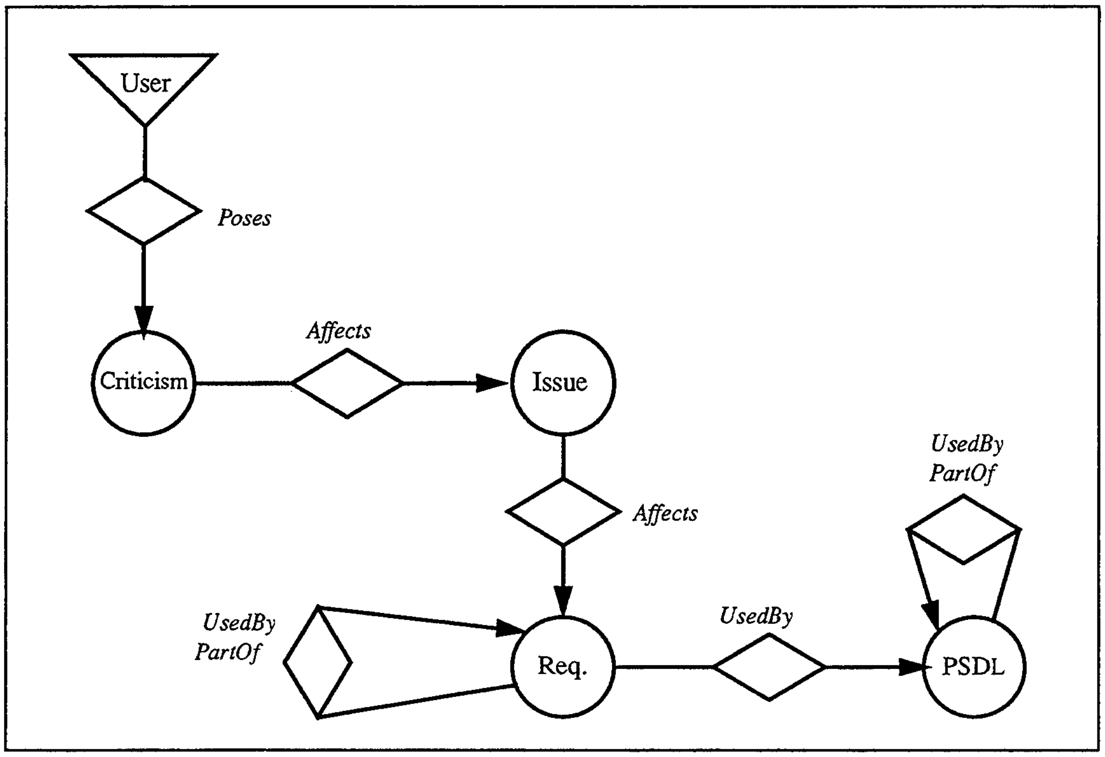


Figure 3.1 Dependency Diagram

1. *PartOf*: connects objects of the same type and represents the decomposition structure of software components or steps. The inverse relation is called *HasParts*.
2. *UsedBy*: links two components of the same subclass or different subclasses. This relation is intended to mean that *Object Y is UsedBy X if the semantics or the implementation of X depends on, is affected by, or uses the semantics of Y.* The inverse relation is called *Uses*.
3. *Primary Input*: links an object to be updated to step that will realize the new version of the object.
4. *Secondary Input*: links an object to the steps that need read-only access to the

object.

5. *Output*: links a step to its output.
6. *Affects*: links a criticism to an issue or an issue to a requirements component that it affects. This relationship is a specialization of and implies a *UsedBy* relationship. *Affects* relations represent dependencies that are explicitly declared by analyst or designer, rather than those calculated from other relationships or from the contents of the components.
7. Poses: links a user to a criticism he poses.

Some of the objects and relationships of our model as given in Figure 3.1 above are similar in interpretation to the IBIS model in which a given set of issues are to be resolved in the presence of different positions justified by arguments. However, the original IBIS model and its extensions implicitly assume that the set of issues are independently specified. In our model, issues are created using criticisms posed by the stake holders. One other basic difference is that in our model issues are related to design artifacts from the highest to the lowest level, thus it spans the whole life cycle of the system under development not only the deliberation process. Some other differences follow:

1. In IBIS model and its extensions, the process stops once an alternative is selected. In our model, an alternative is selected based on an analysis process and a proposed plan is automatically generated to implement the resolution. Within the context of this plan, the consequences of the proposed implementation are exposed for further analysis and adjustments by the management.
2. Our model makes better use of the knowledge captured during the process in evolving requirements and design consistently into new versions that more closely approximate the user real needs.
3. The demonstration process provides a better context for the elicitation process where the demonstrated examples of system behavior stimulate customer reactions and judgements (what you see is what you get).

In Chapter V we will return to the IBIS model where we improve it to alleviate these and other drawbacks. Our improvement provides more representation power to the IBIS model by introducing necessary types and relationships. These new types explicitly represent the outcome of the deliberation process. They allow the representation of alternatives, criteria, group decisions. More importantly, we will combine it with an improved formal technique that we will use to quantify stakeholder judgements.

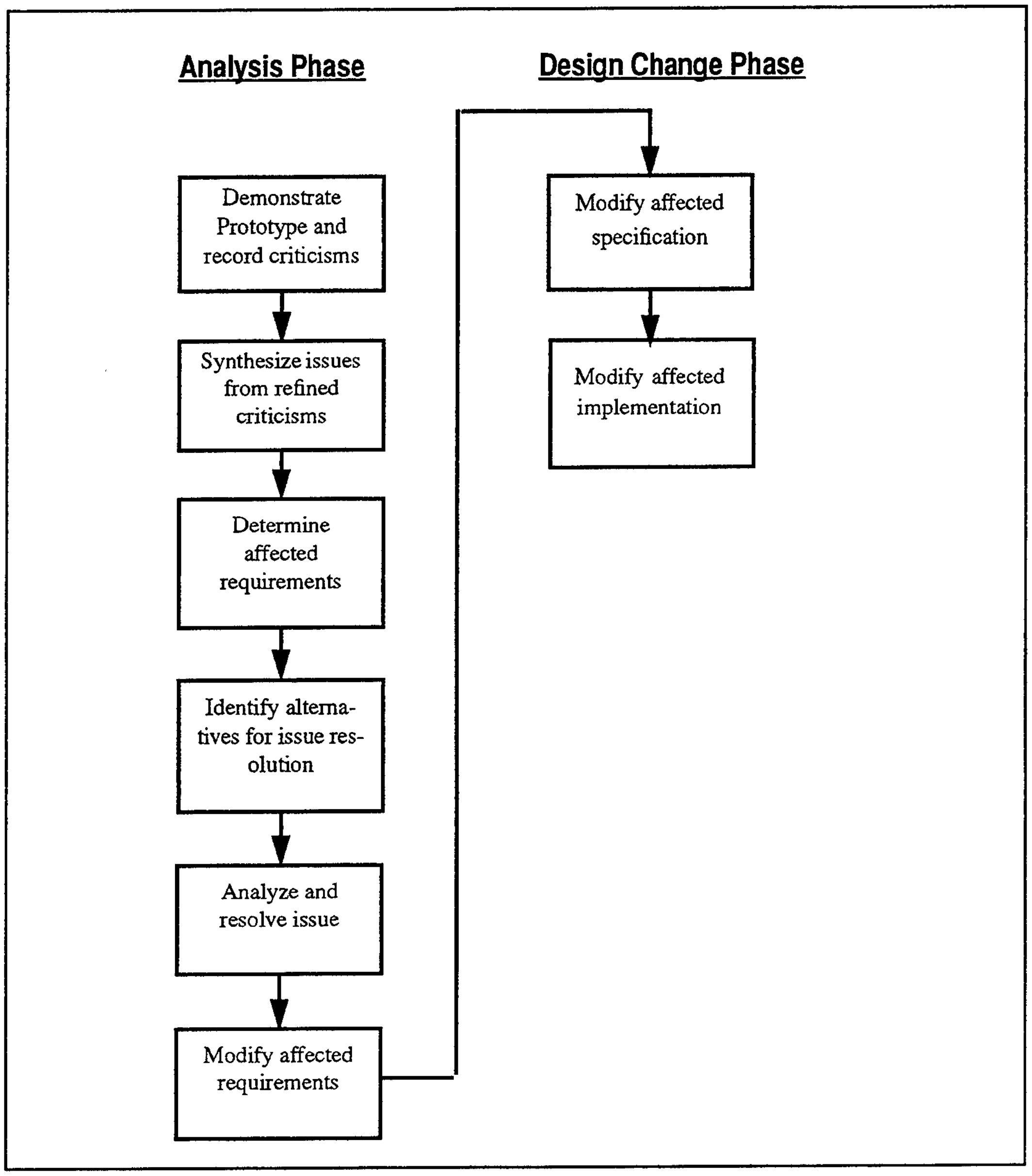
3. Formulation of The Model

Our model is based on a directed graph G (V,L) where the set of vertices has three disjoint subclasses $V = \{H \cup S \cup C\}$ and L is the set of links (edges) of the graph. *H* models the set of persons (users and designers) involved in the process and S models steps with both their variants. The class *C* represents software components and has the following subclasses which have the same attributes and relationships but whose text attributes have semantically different interpretations:

1. P: The entire system (prototype) to be demonstrated.
2. R: The set of criticisms generated during the demonstration of P.
3. I: The set of issues affected or addressed by the criticisms; each issue is either generated during the current demonstration or evolved from one that already exists as a part of the demonstration history.
4. Q: The set of requirement components organized in a hierarchical structure. Each requirement is either newly created during the process or evolved from a previous version.
5. D: The set of the of PSDL components that represent the design structure of the prototype. This subclass includes specification, and implementation structures.
6. T: The set of test scenarios that is used to test the system behavior as perceived by the customers.

4. The Process

The associated process can be viewed as two consecutive phases: analysis and design change phase, as the simplified schematic in Figure 3.2 shows.



Analysis Phase
Design Change Phase
Demonstrate Prototype and record criticisms
Synthesize issues from refined criticisms
Determine affected requirements
Identify alternatives for issue resolution
Analyze and resolve issue
Modify affected requirements
Modify affected specification
Modify affected implementation

Figure 3.2 Process Schematic Diagram

In the analysis phase, the customer reactions to the demonstrated behavior of the prototype are captured and used to synthesize a set of issues to be resolved. The requirement components to be manipulated during the resolution process are determined and the various alternatives available for the resolution are identified. Issues are resolved by choosing among these alternatives. This phase ends by modifying the affected requirement components. In the design change phase, the actual changes in the affected design specifications and implementations are carried out.

Typical model elements affected by the process are illustrated in Figure 3.3 below.

a. Demo Step

The analysis process associated with the demonstration activity proceeds as follows: at the planned date of the demonstration, having the prototype and the test scenarios ready, the prototype is demonstrated in the presence of customers. The result of the demonstration is a set of criticisms posed by the customers in response to the demonstration. In a demonstration analysis substep, these criticisms are recorded, reviewed by the customers for accuracy, and entered into the project database along with amplifying information. The following tasks are associated with this substep:

1. Record the set of the generated criticisms augmented with analysis information such as relation to other criticisms, justifications, agreement with the user goals and constraints, and any elaboration that may clarify the customer needs. However the raw part of the criticism as it is cited by the user is kept separate and preserved intact for future reference.
2. Establish the link between users and the generated criticisms. This link is used to trace contradictions back to conflicts of interest between user groups, to focus negotiations, and guide priorities.
3. Analysis of criticisms for clarity, plausibility, and consistency.

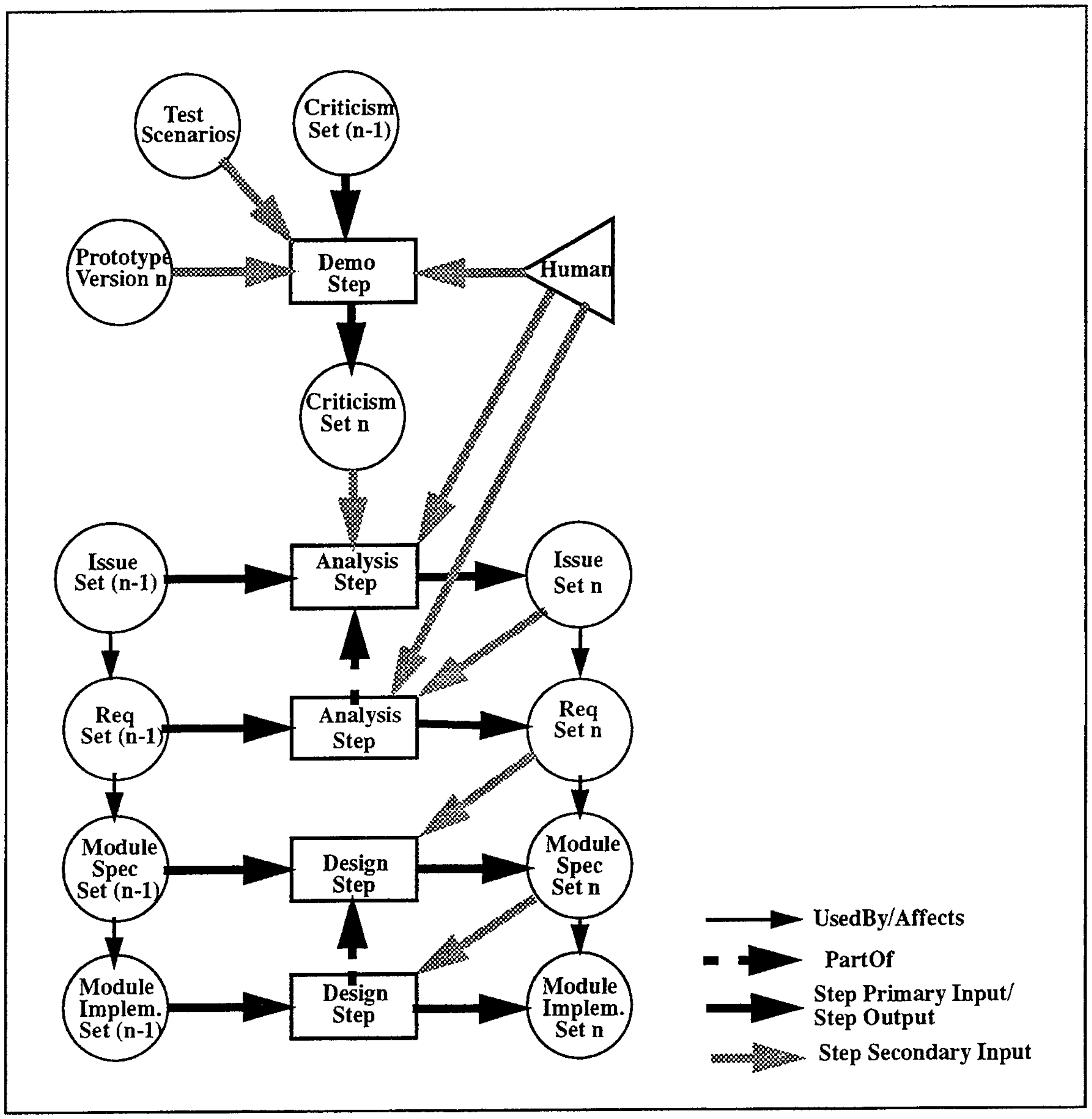


Figure 3.3 Schematic Model of the Analysis Process

b. Issue Analysis Step

When the manager advances the status of the demonstration analysis step to *completed,* an issue analysis top level step is automatically generated. This step works on the set of criticisms recorded at the current demonstration and the set of issues from the previous demonstrations. The main task of this step is evolving to a new set of issues assembled using the generated set of criticisms and the issue set from the previous demonstrations. Within the context of this analysis step the following analysis tasks are identified and resolved:

1. Merging, generalization, and reformulation of similar criticisms.
2. Dividing the set of criticisms into subsets each of which addresses one issue. These subsets are not necessarily disjoint to account for cases where one criticism addresses more than one issue.
3. Manually linking each criticism to the issues from previous demonstration that it addresses (if any).
4. Generalizing or formulating the statement of any of the existing issues (if necessary) to adapt the issue to the subset of criticisms linked to the issue.
5. Generating new issues if any of the criticisms are not linked to any issue.

The output of this step is a refined set of issues to be resolved. To resolve these issues, each is considered separately within the context of an automatically generated analysis substep as part of the requirements analysis top level step.

c. *Requirements Analysis Step*

This top level step is automatically generated at the completion of the above step. The primary input of this step is the set of requirements affected by the synthesized issues. The issues themselves are the secondary input. This top level step has a number of substeps equal to the number of the synthesized issues. Within the context of this step issues are linked to the requirement components they affect in the existing requirements hierarchy and to newly created requirements if needed. Each issue is resolved within the context of one of the substeps. A substep of these is concerned with:

1. Generating new requirements if the issue addresses a missing requirement.
2. Exploring the availability of different alternatives for resolving the issue from the requirement components (either existing or proposed) affected by the issue.

3. Initial mapping of the new affected requirements into the existing design artifacts.
4. Generating relevant information that assists stakeholders in their judgement on the issue resolution.
5. Resolving the issue which includes:
   - The stakeholders debates to choose among alternatives.
   - The group decision as to which alternative(s) to select.
   - A change request that maps the group decision into required changes in the requirements and accordingly in the system design consistently.

All these tasks are explained in detail in Chapter VIII. as part of a case study. The fourth task is explained in detail in Chapter IV. The last task is the subject of Chapter V.

One way of exploring the availability of different alternatives to resolve an issue in our context is by examining the *Affects* link between an issue and the set of requirement components it affects (either existing or proposed). These alternatives can be derived from this set of requirement components (Call it $Q_{affected}$). If this set can be divided into a number of n independent subsets (Call it Q_i), the issue has a number of alternatives equal to n. An issue can be resolved by updating the requirement components of only one of these subsets while keeping the other subsets intact.

This means that according to the value of n, we have two cases:

1. $n = 1$ means that all requirement components affected by the issue are to be updated for resolving the issue; no other options are available.
2. $n > 1$ means that the set of the affected requirement components can be divided into subsets where the issue can be resolved by modifying the requirement components in only one of them. Each of these subsets represents the basis of an alternative to resolve the issue.

If r_i are the requirement components in Q_{affected} and the latter's cardinality is m, the first case implies that all $(r_1 \wedge r_2 \wedge \ldots \wedge r_m)$ must be updated together to resolve the issue.

The second case implies that any Q_i can be updated to resolve the issue, where:

a. $Q_i = \{r_1, r_2, .., r_j\}$, and

b. $1 \le j \le Cardinality(Q_{affected})$, and

c. $\bigcup Q_i = Q_{Affected}$.

Some or all of the requirement components in Q_{affected} may already exist as part of the requirements hierarchy. Others may be newly proposed components accounting for e.g., a missing functionality.

The decision making in the first case deals with selecting the right statement of each affected component. If differences arise, a formal debate is conducted to reach to an agreement. The debate among stakeholders is concerned mainly with what the updated requirements would be and the impacts of the proposed changes on their goals. The detail of such a process is given in Chapter V.

The second case requires more work and analysis. Each identified subset bears the basis of an alternative candidate to resolve the issue. The choice among these alternatives will be explored in Chapter V within the context of a formal technique.

For example, in demonstrating a prototype that models a generic C^3I system [46], one of the criticisms posed is concerned with timing error from the PSDL module representing weapon status, meaning that the maximum execution time (MET) of this module (PDSL operator) is violated. The issue synthesized from this criticism is “Timing constraints violation”. To resolve this issue, two alternatives are available: changing the timing requirements for all time-critical modules with the intention of increasing the MET of the weapon status module and hence decreasing the METs of the other modules, or changing the hardware speed requirement of the real system to increase the processor speed. The first alternative affects a subset of requirement components that include the timing requirements for the weapon status and other subsystems (many requirement components). The second alternative affects only one requirement component: the one that specifies the hardware requirement of the intended system. Choosing that alternative implies that all requirement components in the associated requirement subset will be manipulated to implement the chosen resolution. Although the second alternative takes less effort, the preference process is done within the context of other deciding factors. For example it may be impossible due to budget constraints to upgrade the hardware architecture to provide the desired speed. In Chapter VIII we provide a more detailed example.

To assist stakeholders in the selection process, the decision support mechanism gathers relevant information related to each alternative. The dependency graph of the current design is used to support this process. This information is then used to attribute each alternative. Values here are rough estimates and are only used as indicators to guide the judgement of the stakeholders.

The information is gathered by traversing the portion of the dependency graph from the affected subset of requirement components down to the leaves of the dependency graph (implementation modules), following *UsedBy* and *PartOf* links. The kinds of information gathered are:

1. The number of the specification modules (PSDL operators) affected (and hence the number of implementation modules).
2. If any additional specification modules are required (and hence any additional implementation modules).
3. The availability of designers with the required expertise level and field to carry out

the work required by the resolution.

4. Optionally, the availability of reusable components in the software base for any new implementation modules needed as a result of the proposed change may be explored.

d. Design Change Step

The analysis phase ends by approving one or more alternatives for resolving each issue. The result is a set of approved change requests to be prototyped in the design change phase. Each of these changes implies a hierarchy of design steps through a chain of induced steps from the requirements down to the implementation levels, following the *UsedBy* and *PartOf* relations. At this stage, for each of the approved change requests, the affected subset of requirement components are known for each issue. This induces a set of implied design steps, one for each affected component.

The change propagation from the affected requirements components down into the design hierarchy is performed within the context of an automatically created set of steps. Each of these steps takes one of the specification modules linked to an affected requirements component as a primary input and the requirement component as a secondary input and outputs a modified version of the specification module. Each of these steps can spawn another set of induced steps that modify the affected implementation modules. Each such induced step takes one of the affected implementation modules as a primary input and the affecting specification module as a secondary input and outputs a modified version of the implementation module.

IV. DECISION SUPPORT MECHANISM BASED ON THE MODEL

The kind of decision support and automation provided assists managers and analysts in making decisions but does not replace them. Managers can always override the generated values, change status of the different activities, add or remove relationship instances, and choose options rejected by the decision support mechanism. For example, in most cases affected modules are calculated successfully from the dependency graph using the existing relationships. However, initially manual intervention is needed to build these relationships. Also because of the process dynamics and the ongoing concurrent changes, components are added (deleted) to (from) the system which requires the intervention of managers. They first review the result of the automatic computation and then adjust the computation result (if necessary) to reflect these changes. This makes the process always under the control of the project managers and the analysts for human judgement and further considerations.

A. TYPES OF DECISION SUPPORT PROVIDED

The types of decision support provided span a wide range of specific tasks and activities. The automatic support and reasoning facility the system provides is based on a set of rules to be used in automatically deciding the right action to be taken. Actions are concerned primarily with the following tasks and activities:

1. Computing or inferring relationships from given ones.
2. Automatically propagating change consequences to other parts of the system.
3. Assisting in establishing consistent planning and control.
4. Coordinating task implementation.
5. Controlling status of the analysis and design activities.
6. Providing support for monitoring and adjusting the execution of the plan.

This section outlines types of decision support provided based on the conceptual model and the associated set of rules and constraints that governs the process. We use the notations given in Table 4.1 to present some of the rules:

Notation	Meaning
$s/S/s_i$	Analysis or design step.
$r/R/r_i$	Criticism.
$i/I/i_i$	Issue.
$q/Q/q_i$	Requirements component.
$p/P/p_i$	PSDL component.
$v/V/v_i$	Version of any component.

Table 4.1: Notations

1. Automatic Generation of Analysis and Design Activities.

Approval of an analysis or a design step that considers a component for change triggers an automatic process that computes the other affected components, and generates a substep to consider each of these affected components. The generated substeps inherit the parent's step "Approved" state. Managers can intervene to adjust the result of this mechanical process by adding to or deleting from the affected components set to account for cases of missing or existing relationships that the automatic process did not consider. The manual adjustment also requires creating a substep for each newly added affected component and abandoning all substeps corresponding to all deleted affected components. Depending on the status of the parent step S, the rules associated with this mechanical process are [69]:

1. If S is "approved" then add the corresponding substep.
2. If S is "scheduled" then:
   - include the effects mentioned in 1 above, and
   - modify the dependency graph to reflect these changes, and

- recalculate the schedule according to the modified graph.

3. If S is "assigned" then:
   - include all the effects mentioned in 2 above, and
   - suspend any assigned steps that become dependent on any of newly added steps, and
   - assign any of the steps that become ready.

2. Dependencies computation

This includes computing or inferring relationships from given ones and updating these relationships as more information becomes available. Examples of such computations are as follows:

1. Computing the set of components affected by the change to the primary input component of a step.
2. Computing the set of secondary input components of a step.
3. Computing the set of usedBy components of a step.

Rules that govern this computation are as follows:

a. Affects Rule

For any two versions *v* and *v1*, if *v* affects *v1* then *v* is used by *v1*.

Rule:

ALL(v v1: V:: (v affects v1) $\Rightarrow$ *(v usedBy v1))*

This rule is used to derive and compute the *usedBy* relation from the *affects* relation. This rule is illustrated graphically in Figure 4.1. For example *v* may be an issue that affects a requirement component *v1*. If we are to modify *v1* by a step *s*, we need a read-only access to *v*. Hence *v* must be one of the secondary inputs to *s*. But since the automatic computation of secondary inputs is based on the *usedBy* relation, changing *affects* relation into *usedBy* relation by the above rule is necessary and valid transformation.

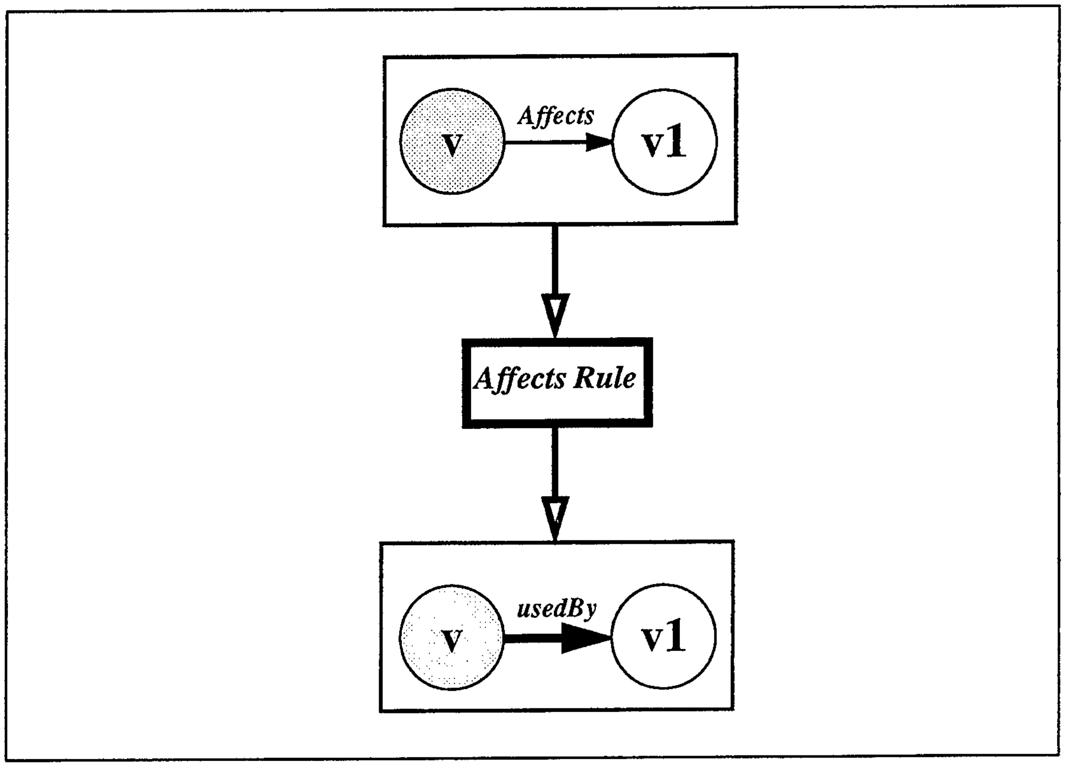


Figure 4.1 Graphical Representation of the *Affects* Rule

b. UsedBy Rule

For any two versions *v*, *v1* and a step *s*, if *v* is one of the secondary inputs to *s* and *v1* is one of the outputs of *s* then *v* is used by *v1*.

<u>Rule:</u>

ALL(s: S, v v1: V:: (v ∈ secondary_input(s)) & completed(s) & (v1 ∈ output(s)) ⇒ (v usedBy v1))

This rule adds to the above one in computing the *usedBy* relations among the different elements of the system. These elements can be objects of different type components or can be objects of the same type component as the next rule shows. This rule is used mostly in the initial building of the *usedBy* relation. *usedBy* is built from:

- The *Affects* Relation as explained above.

- The *PartOf* relation as we explain in the next rule.
- The secondary input set of a step.

The last case occurs in a situation where for, example, a manager reviews the task associated with a step. He may see that a read-only copy of some version is needed to complete the step task. Therefore he adds this version as one of the secondary inputs to the step. This version is not originally linked to the primary input of the step by *usedBy* relation. Otherwise it would have been computed by the automatic process as one of the secondary input to the step. That is why this rule is necessary to account for such situations.

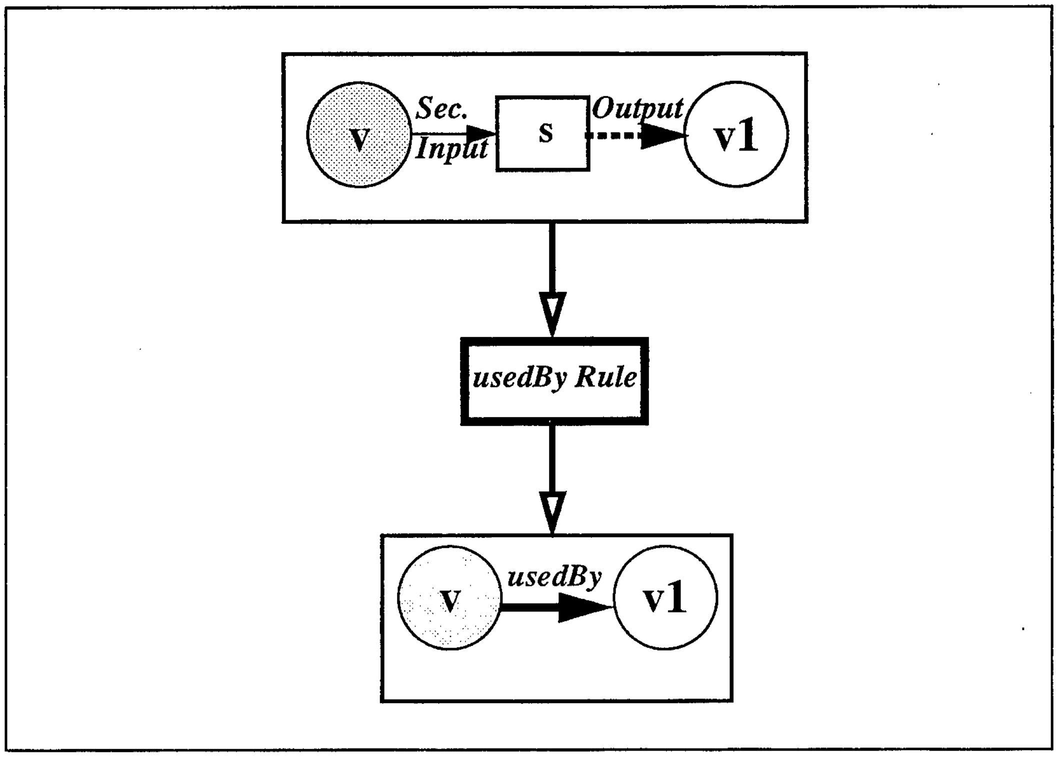


Figure 4.2 Graphical Representation of the *usedBy* Rule

c. *PartOf Rule*

For any two requirement components *q* and *q1*, if *q1* is *PartOf* *q* then *q* is *usedBy* *q1*.

Rule:

ALL(q q1 : Q :: (q1 PartOf q) ⇒ (q usedBy q1))

This rule has another variant that works on the PSDL hierarchy. In [69] it was assumed that in the case of PSDL specification modules, children are used by their parents. More specifically, the usedBy relation links a submodule specification to its own implementation as well as to the specification and implementation of its parent module. This assumption is not always correct. Refer to section A.2.f of this Chapter for more elaboration.

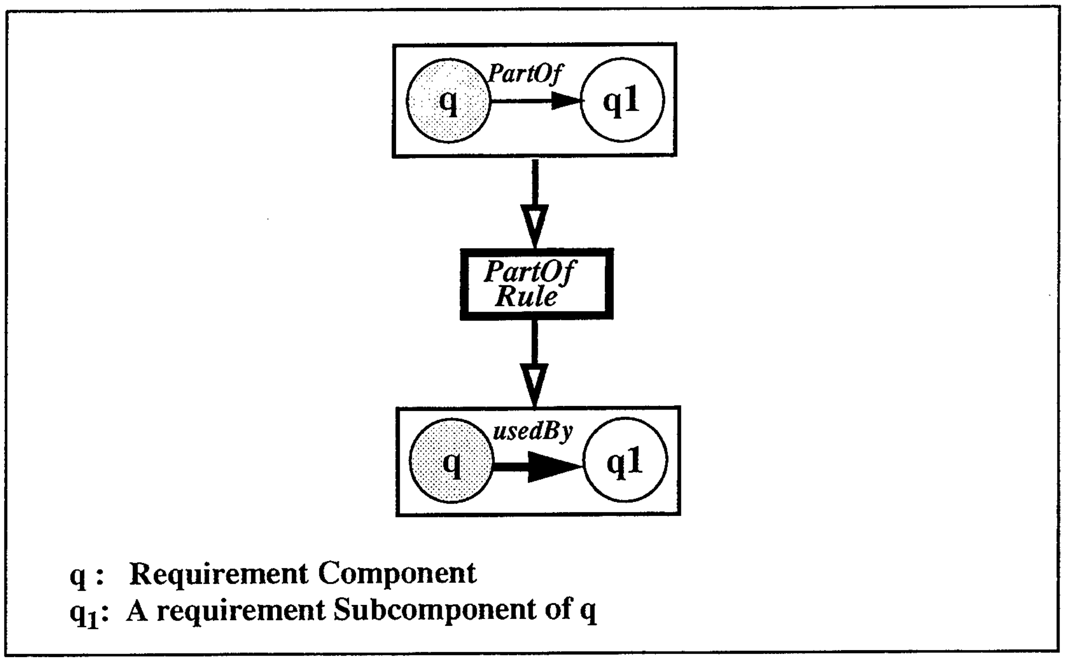

q
PartOf
q1
PartOf Rule
q
usedBy
q1
q : Requirement Component
q1: A requirement Subcomponent of q

Figure 4.3 Graphical Representation of the *PartOf* Rule

d. Affected Components Rule

For any two versions *v1* and *v2* and a step *s*, if *v1* is a primary input to s and *v1* is *usedBy v2*, then *v2* belongs to the set of the affected components of *s*.

Rule:

ALL(v1 v2: V, s: S:: (v1 ∈ PrimaryInput(s)) & (v1 usedBy v2) ⇒ (v2 ∈ AffectedComponents(s)).

All of the rules presented earlier in this section are building rules. They are used to incrementally establish relationships among the system parts. These rules directly serve the evolution process by mechanically computing the impact of a change in one part of the system on the other parts.

Since all relationships that bear the semantic of *affects* (e.g., *Affects* and some *PartOf* instances) are transformed to *usedBy* relationships, this rule is enough to compute the consequences of changes. Of course combined with human interventions and adjustments. Further, since *usedBy* relation implies dependence of the *user* on the *usee*, this relation can be used in computing affected components due to a proposed change in another component.

e. Secondary Input Rule

All the components affecting a version of another component updated by a step are default secondary inputs to the step. These defaults can be manually adjusted if needed. Since a version v that affects another version v1 implies that v is used by v1 (by rule a), it is redundant to specify a separate rule for computing secondary inputs from the *usedBy* relationships because it is automatically deduced anyway.

The line of reasoning behind the sufficiency and validity of this rule is similar to that given to the rule in d above.

Rule:

ALL(s: S, v v1: V:: (v1 ∈ primary_input(s)) & (v affects v1) ⇒ (v ∈ secondary_input(s)))

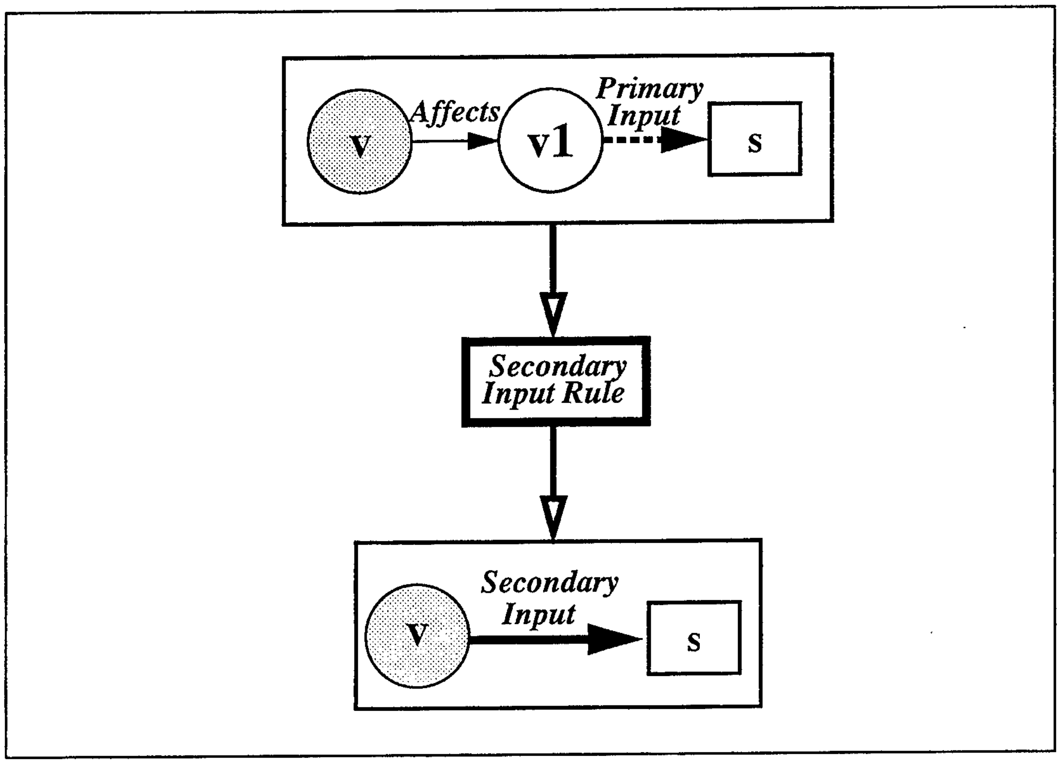


Figure 4.4 Graphical Representation of the *Secondary Input* Rule

Some of the rules presented in this subsection are introduced for the first time. Others are derived from and modify similar ideas extracted from previous works [47], [69]. The *Affects* rule is introduced for the first time because of our extending the graph model to incorporate requirements analysis. This required the introduction of the *Affects* relation to tie together some parts of the system (see Chapter III).

The *usedBy* was introduced as a relationship in the above previous works. Secondary inputs and affected modules were defined in terms of this relation. In the previous works there was the implicit assumption that all the instances of this relation is derived automatically. Our work formalizes this relation and extends the automatic process to not only using this relation in deriving the secondary inputs and affected modules, but also to build this relation as explained above. The semantic of the *usedBy* rule introduced in b was not accounted for in the previous works.

The part of the *PartOf* rule introduced in c that establishes the *usedBy* relation in the requirements hierarchy is introduced for the first time. We also formalized the concept of the affected components in the rule given in d.

We extended the process that computes the secondary inputs. This is introduced in the rule given in e. Instead of basing the computation of the secondary inputs on the *usedBy* relation only, our work bases it on the *Affects* relation. Firstly, because we have the *Affects* relation explicitly modeled. Secondly, the *Affects* relation implies a *usedBy* relation.

f. UsedBy Direction in the PSDL Hierarchy

The direction of the *UsedBy* relationship in the PSDL hierarchy is complicated by a few facts that should be considered carefully when dealing with that relation either to automate its generation or to manually adjust the computed values. By *direction* here we mean the domain and the co-domain of the relation which is represented graphically by an arrow from the former to the latter. These facts are:

1. In the initial creation of PSDL components a parent *Spec* is *UsedBy* its *implementation,* and this *implementation* is *UsedBy* each *child Spec*. This is the default direction of the *UsedBy* relation. Figure 4.5 depicts this fact.
2. Changes can make dependencies in the reverse direction. This situation is illustrated in Figure 4.6. The default direction of *UsedBy* is reversed when a step modifies I0. In this case SA0 and SB0 are needed as secondary inputs to this step.

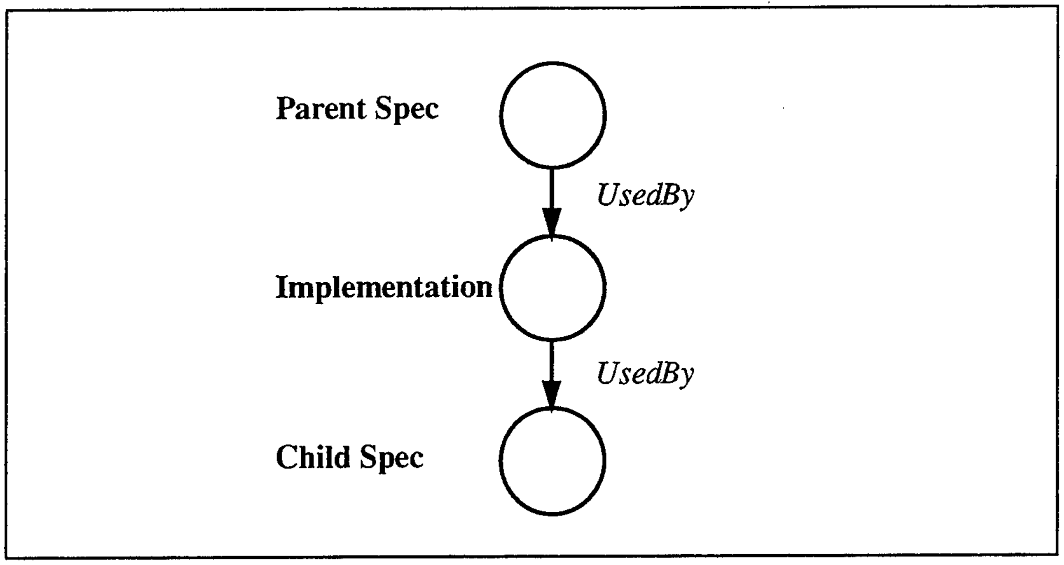


Figure 4.5 Direction of *UsedBy* for Initial Creation

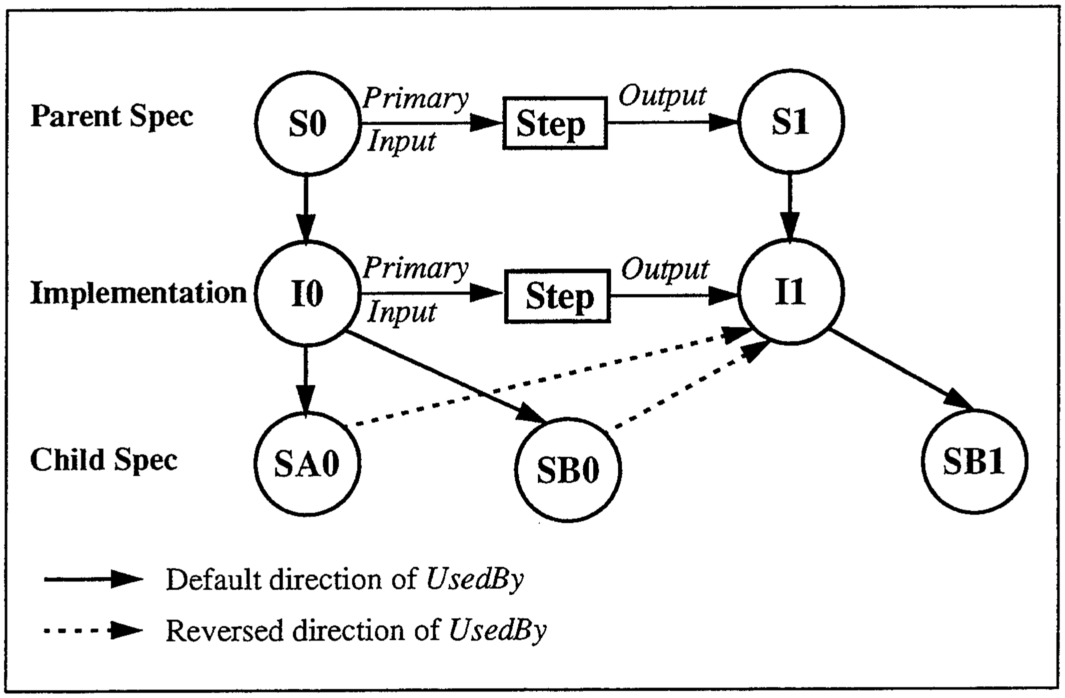


Figure 4.6 Changes can Reverse the Direction of *UsedBy*

3. Alternatives Generation and Evaluation Support

This type of support assists in the analysis phase of the process to:

1. Identify the set of alternatives available for resolving an open issue.
2. Gather relevant information that assists stakeholders on their judgements on the available alternatives.
3. Support the process of the independent judgement of each stakeholder.
4. Support the process of combining the individual judgements into one group judgement and final decision.

These tasks are performed within the context of a top level analysis step generated automatically for resolving an issue. Two substeps are generated as part of the latter step. The first deals with the determination of the available alternatives and generating relevant information. As part of this substep, the affected parts of the system design are exposed and a proposed plan to modify these parts is generated. This plan takes the form of a series of induced steps. The second substep deals with the kind of support required by steps 3. and 4. above. Stakeholders are directly involved in the tasks associated with this substep. We elaborate the detailed process related to this activity in Chapter V.

The process ends by selecting one or more alternative(s) according to the combined judgement of the stakeholders. The manager or analyst approves the step associated with the chosen alternative(s). Accordingly, the system propagates the approval to all substeps related to the selected alternative, and rejects (abandons) other alternative's activities. Following are the detailed procedures in support for generating and evaluating alternatives.

a. Alternative Generation

Once the set of the requirement components affected by each issue is determined, it is then manually linked to the issue, This set is also partitioned (if applicable) into independent subsets Q_i. Each Q_i can be manipulated independently from other subsets to resolve the issue. This partitioning is based on the technical experience of the analysts. An issue can be resolved by changing existing requirement components, creating new ones, or combination of both.

If the manager (analyst) then issues the *GenerateAlternatives* command to the system, the support facility creates a number of alternatives a_i each corresponds to one independent subset of the affected requirement components. Each alternative is added to the list of alternatives for resolving that issue and is marked *tentative*. Tentative here indicates that this alternative is not necessarily the one to be used in resolving the issue.

Within this activity, the system gathers relevant information related to each of these candidate alternatives. This information is used along with others to guide stakeholders in the selection process. Automation abstraction in support of this process is depicted by the algorithm in Figure 4.7 given for one open issue *I*.

```
Algorithm GenerateAlternatives (I: input Issue)
Input: An issue I
Output: A set of tentative alternatives to resolve I with a proposed top
level step associated with each alternative.
Begin
    If I affects more than one independent requirements
    subset $Q_k$ then
        For each $Q_k$ do
            Create an alternative $a_i$
            $a_i$.requirements_set := $Q_k$
            $a_i$.status := tentative
            Add $a_i$ to I.alternatives_list
        Generate a step S in the proposed state such that
            S.primary_input := $Q_k$
            S.secondry_input := I
        end do
    end if
end GenerateAlternatives.
```

Figure 4.7 Alternatives Generation Algorithm

This algorithm abstracts only the main tasks involved in the alternatives generation support. Many details need to be worked out for actual implementation. The algorithm takes as an input parameter one issue I to be resolved. This parameter is most likely a reference to a structured object of type *issue*. Some of this object attributes are already assigned values (see "Other Types" on page 152) for the attributes). An example is the requirement components affected by the issue. Some other attributes are to be assigned values by the algorithm. An example is the list of alternatives worked out and assigned by the algorithm. Before the algorithm can work on it, the issue should have gone through some analysis stages. The set of the requirement components affected by I must have been determined. This set also must have been partitioned into independent subsets (if applicable). Alternatively the affected set could be partitioned within the scope of the algorithm. In the latter case one possible scenario is as follows:

1. The system presents the issue to be resolved to the analyst along with the affected requirement components set $Q_{affected}$.
2. The analyst reviews the $Q_{affected}$ set and chooses *partition* (if applicable). New requirement components can be proposed here too.
3. If the analyst chooses *partition*, he has to specify the requirement components in each partition. The system makes this task easy. Except for the mental task of determining that there is a feasible partitioning, all the analyst has to do is clicking the mouse buttons to perform the actual partitioning.
4. For each partition the system creates an alternative. The content of an alternative is the requirement subset in the partition. The system assigns the value *tentative* to the issue status.

b. Alternatives Evaluation

The decision support mechanism assists in the alternatives evaluation process by providing the following:

1. Gathering information relevant to each alternatives.
2. Assigning values to the attributes of the alternatives based on the gathered information.
3. Assisting stakeholders in weighting the criteria of judgements agreed upon by them.

4. Assisting in expressing the preference of each stakeholder for alternatives.
5. Assisting in combining individual stakeholder preferences into one that expresses the preference of the group. A group decision is based on the latter.
6. Assisting in generating a drafted change request that expresses the group decision.
7. Providing the capability to dynamically change measures (or their values) used in the selection process.

Chapter V provides detailed elaboration on a methodology to be used in implementing steps 3-6 above. In the rest of this section we concentrate on the process of gathering the required relevant information in support of the selection process.

4. Relevant Information

The relevant information gathered in support of the selection process is made available to stakeholders. It gives them a kind of aspects that characterize each of the debated alternatives. This information also puts the fingers of the technical team on the expected consequences of implementing a specific alternative. However, the support mechanism offers only advice which can be overridden by stakeholders.

The decision support mechanism uses the dependency graph and the design database to gather the required information by following the relationship links of the current state of the graph. In the following we present this kind of information which is concerned with the new PSDL components required to resolve an issue as well as the existing PSDL components to be manipulated for resolving the same issue. The computation is performed for every alternative that can be used to resolve the issue. The gathered information also includes the availability of designers of the required expertise field and level to carry out the implementation effort of the selected alternative(s).

a. An Estimate of the Issue Resolution Effort

This piece of information represents an approximate number of the new PSDL components (specification and implementation) to be designed and implemented as a result of selecting an alternative to resolve an issue. The figure is used as a quantitative measure indicating the cost of implementing an issue resolution using a specific alternative.

This measure is crucial, for example, to catch up with a deadline. Of course minimizing effort should not have a negative impact on satisfying customer needs. The information gathered assists in identifying the effort associated with the implementation of each alternative. Alternatives can then be ordered by this measure for stakeholders to choose according to which if minimizing effort is a major concern.

The ordering is based on the number of the new PSDL modules ***(P_i_New)*** required. This figure is approximated by the number of the requirements components in the requirements subset of the alternative which are not currently linked to any PSDL module by *uses* relationship. This rule of computation bears the implicit assumption of a one-to-one mapping between the PSDL and the involved requirement components which is not always true. As an indicator, this simplification is acceptable, however. Further, the computation validity is subject always to human reviews and adjustments.

The algorithm in Figure 4.8 below depicts the computation actions to determine *P_i_New*.

```
Algorithm Compute $P_i$_New(I, a)
Input: an issue I and alternative a such that $a \in I \cdot alternatives$
Output: $P_i$_New (the number of the new PSDL modules required to
        resolve I using a).
Begin
Initialize $P_i$_New to 0
For all q such that $q \in a \cdot requirements$ do
        For all components c such that $c \in (q \cdot UsedBy)$ do
            If c.type = PSDL then /* q can be linked by usedBy relation
                                        to another requirements component */

                exit    /* q is Linked to at least one existing PSDL
                            component */
            else
                Increment Pi_New
            end if
        end do
end do
Return $P_i$_New
end Compute $P_i$_New
```

Figure 4.8 Algorithm for Computing the New Required PSDL Components

b. An Estimate of the Issue Resolution Complexity

The kind of information gathered here supports the evaluation of an alternative in terms of complexity. Again it is an indicator only that can be overridden by human judgement. This information answers the question of how complex is resolving an issue by taking the course of actions proposed by a specific alternative. The answer is extracted from the effect on the different parts of the prototype design as a consequence of implementing the resolution using that alternative.

This effect is quantified by the number of the affected PSDL modules assuming that as this number increases, so does the complexity of implementing the resolution. This is because firstly, modifying many PSDL components indicates that much care should be taken to analyze the impacts of these changes. Secondly, the wider the changes spread, the more things goes out of sync. For example, timing constraints adjustments may lead to rescheduling under the potential of finding no feasible schedule. Thirdly, the wider the span of the proposed changes, the more probable inconsistencies are created and the more expensive to fix.

The decision support mechanism extracts the required information from the dependency graph by computing the number of PSDL components ***(P_i_Affected)*** that are affected by selecting the subject alternative.This computation is performed by counting the number of PSDL components linked to the requirements subset to the lowest level in the dependency graph. The *usedBy* relation is used in this kind of computation.

In this case the alternatives available for resolving an issue are presented to the decision maker ordered by that measure where the one with the minimum affected components comes up first.

The algorithm given in Figure 4.9 illustrates this computation process. In this computation there is the danger of over counting the required PSDL modules if the ***usedBy*** relation is only used in the computation. This is because the ***usedBy*** relation not only links requirement components to the PSDL components, but also transitively links requirement and PSDL components to their parent components. Figure 4.10 illustrates this problem. In this figure a fragment of a system prototype showing the links between the fragment of the requirements hierarchy and the corresponding fragment of the PSDL modules hierarchy.

For simplicity we assume a one-to-one mapping between requirement components and the PSDL modules. As one can see from the figure, the number of the PSDL components linked by *usedBy* to the requirements fragment can be overcounted. This happens because m1.1 is counted twice as follows:

(1) $q_1 \rightarrow m_1 \rightarrow m_{1.1}$ *(3)* $q_{1\cdot 1} \rightarrow m_{1\cdot 1}$

```
Algorithm Compute P_i_Affected(I, a)
Input: an issue I and an alternative a such that $a \in I \cdot alternatives$
Output: P_i_Affected (the number of the PSDL modules affected by
    resolving I using a.
    Function usedBySet(S, c)
    /* Finds components that uses c and inserts their IDs into and returns
        the set S */
    Input: A set S and a component c
    Output: The Set S added to it the IDs of the components that use c
    Begin
        For all components w such that $w \in (c \cdot UsedBy)$ do
            If w.type = PSDL then
                Insert w.componetID into S
                usedBySet(S, w)   /* Recursive call */
            end if
        end do
        return S
    end -- usedBySet

Begin -- Compute Pi_Affected
Initialize P_i_Affected to 0
Initialize Set S                    /* Empty it */
For all q such that $q \in a \cdot requirements$ do
    usedBySet(S, q)
end do
P_i_Affected = Cardinality(S)
Return P_i_Affected
end Compute P_i_Affected
```

Figure 4.9 An Algorithm for Computing *P_i_Affected*

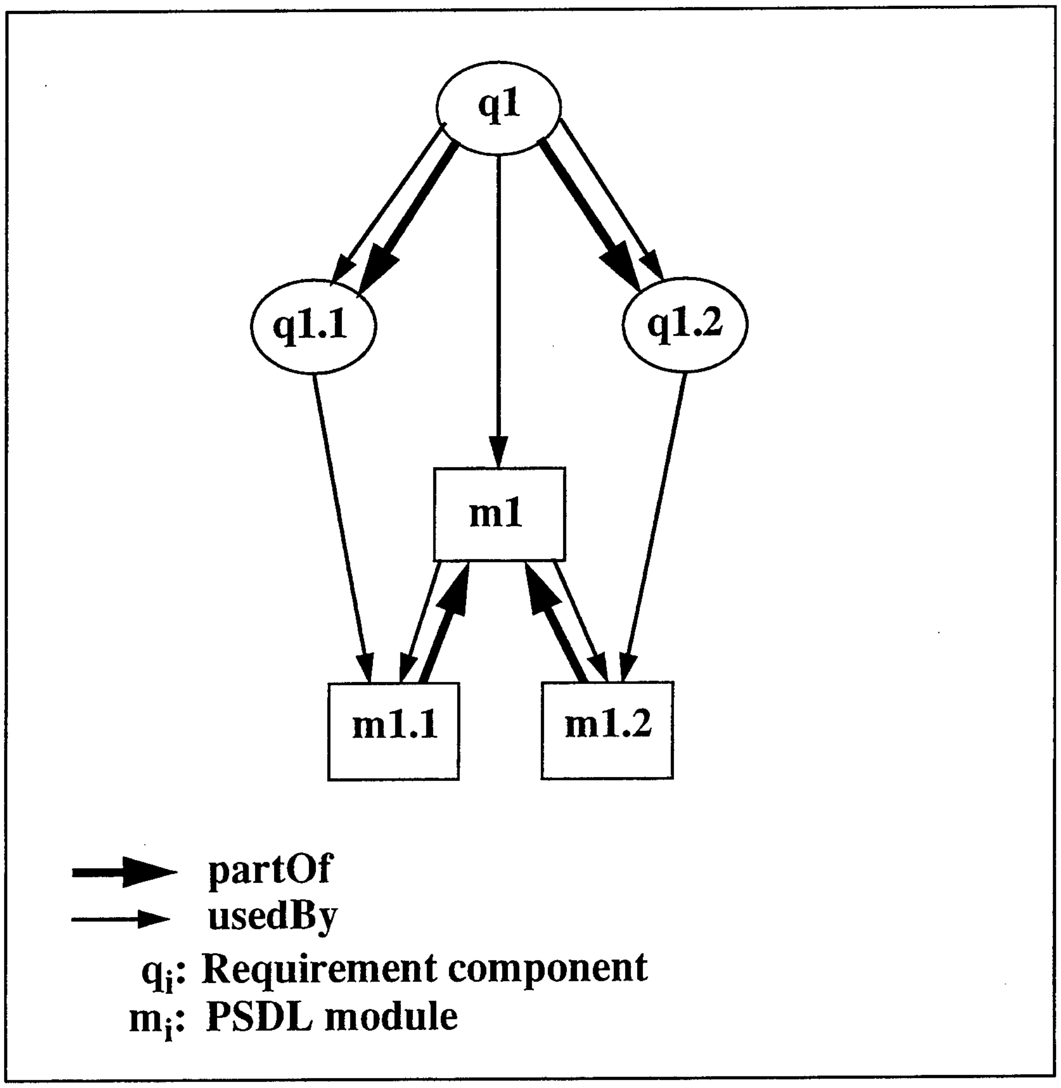


Figure 4.10 Fragment PSDL and Requirements Hierarchy

This problem can be solved by introducing a boolean flag *visited* initialized to *False* and set to *True* when a PSDL component is counted. The flag is checked before counting a PSDL component. If it has the value *True*, the component is not counted. Otherwise it is counted and the flag is reset back to *True*.

Another, perhaps easier, solution is scanning the part of the dependency graph related to the input alternatives by following the *usedBy* relation and inserting the component ID of each encountered PSDL component in a temporary set. Since a component ID is unique and a set data structure does not allow duplicate elements, the ID of each PSDL component affected by implementing the alternative is inserted once. The count of these components is given by the cardinality of the set. This is the solution we employ in the above algorithm using the function *usedBySet.*

B. OTHER TYPES OF DECISION SUPPORT

Other types of decision support provided include supporting the tasks of teamwork, scheduling, assignment of default values, propagating inherited properties, replanning, and task serialization. Following is a brief discussion of some these support functions.

1. Teamwork Support

In support of teamwork, the system identifies tasks that can be executed concurrently by members of the design team. When a designer whose expertise field and level match the task requirements is available, the step is assigned to him and an automatic transition from *scheduled* to *assigned* occurs. Identification of concurrent tasks is computed partially from the dependency graph by considering the relationships between the graph parts. If two steps are to modify two components and the second component depends on the first, the task of modifying the second component can not be performed concurrently with the task of the first one. The precedence among tasks associated with analysis or design steps is effectively an acyclic directed graph G = (S, E) such that the following constraint holds:

Rule1:

$ALL(s1\ s2: S:: ((s1, s2) \in E) \Rightarrow (s1\ Precedes\ s2))$

Most of the precedence relations are calculated mechanically with possible manual adjustments by managers. The following rule is a typical constraint used in precedence computation.

If *s1* is a step whose primary input is the version *v1*, and *s2* is another step whose primary input is *v2* such that *v1* is used by *v2*, then the task associated with *s1* precedes that associated with *s2*. In other words *s2* can not be assigned to a designer until *s1* completes. Note that this constraint is also necessary because the designer assigned the task associated with *s2* will need a read-only access to the output of *s1*.

Rule2:

ALL(s1 s2: S, v1 v2: V:: (v1 $\in$ *primary_input(s1)) & (v2* $\in$ *primary_input(s) & (v1 usedBy v2)* $\Rightarrow$ *(s1 Precedes s2))*

Potential for teamwork may also be available in higher than design level. In the process of resolving issues, resolution of two issues can proceed concurrently if no interdependence exists between them. Interdependence among issues is determined from the set of requirement components each issue affects. For example if issue I_1 affects the set of requirements Q_1 and issue I_2 affects Q_2, then for I_1 to be independent on I_2, the intersection of Q_1 and Q_2 must be empty. This can be formally expressed as:

Rule3:

ALL(i1 i2: I, q1 q2: Q:: (i1 independent_on i2) $\Leftrightarrow$

((i1 affects q1) $\wedge$ *(i2 affects q2)* $\Rightarrow$ $(q1 \neq q2)$*))*

This can represent the basis of high level teamwork support. Issues can be resolved concurrently. After determining the requirement components affected by each issue, a check can be made to identify independent issues. A subteam can then work independently (and possibly concurrently) to complete the analysis tasks of the issue and any consequent changes in the design and implementation. The teamwork support in the context of the first two rules can be employed to support the concurrent work within the scope of the subteam assigned the issue.

2. Scheduling Support

The decision support mechanism enforces the timing and priority constraints of tasks within a schedule. The timing constraint of a step is specified in terms of two parameters: *deadline* and the *estimated duration.* The deadline of a step is the time by which the step must be completed according to customer restrictions or manager's resource planning. The estimated duration is a management estimate of the time needed to perform the step. The values are assigned manually by managers and are monitored by the system during the execution of the schedule. If some constraint is violated, the manager is warned of the situation to intervene and make a corrective action.

Scheduling support has to cope with the dynamics of the scheduling process. This provision also supports incremental replanning as additional information becomes available. Precedence among tasks as well as deadlines and priorities can change dynamically as new steps are scheduled. In response to these changes, the system responds as follows[69]:

1. If S is "scheduled" then:
   - modify the dependency graph to reflect these changes, and
   - recalculate the schedule according to the modified graph.
2. If S is "assigned" then:
   - include all the effects mentioned in 2 above, and
   - suspend any assigned steps that become dependent on any of newly added steps, and
   - assign any of the steps that become ready.

3. Propagating Inherited Properties

In order to preserve the integrity of the whole system under development and make changes conform to a consistent plan, a substep transitions into its parent's status following the approval of the parent. When a step is approved, the system automatically creates a substep for each affected component. The system propagates the parent's "approved" status and any following transitions made by that parent to the created substeps. Following are some constraints enforced by the system in support of the automated control of steps transitions.

1. When a step changes from the "approved" state to the "scheduled" state all of its substeps automatically inherit this transition.
2. When a step is rolled-back from the "scheduled" or "assigned" state to the "approved" state all of its sub-steps automatically inherit the same transition.
3. When a step changes to the "abandoned" state all of its sub-steps automatically inherit the same transition.
4. When a new substep is created, it inherits the same state as its parent step and inherits all version bindings associated with the parent step.

This controlled transition works also up to propagate substeps status to the parent step as in the following two cases. Notice that a parent step makes the transition after ***all*** its substeps make the same transition.

1. A step automatically changes from the "assigned" state to the "completed" state when all of its sub-steps have done so.
2. A step automatically changes to the "abandoned" state when all of its sub-steps have done so.

4. Structuring Support

The basis of the automated decision support provided is derived from two main sources: the set of rules and constraints associated with the support functions and the structuring of the process imposed by the conceptual model represented by the dependency graph with its different components and relationships. The kind of structure enforced by the model assists in exposing and recording the process knowledge of ill-structured problems in general. According to Joanne Linnerooth [39], decision makers are better served by creative help in structuring the problem at hand. This structure should also provide a way of representing decision knowledge that makes it easily understood, updated, and actively used by participants in the decision-making process [85]. Our model provides a well structured representation for:

- Human individuals involved in the process either from the customer or development team side.
- Criticisms generated in response to the system demonstration.
- Issues synthesized from criticisms and linked to the affected requirements.
- Requirements components structured hierarchically linked or to be linked to specification modules.
- PSDL specification modules linked to implementation modules.
- Implementation modules.
- Analysis and design activities linked to the components they modify.

The links in the graph tie the model elements with different relationships that are used in part for providing the automated support. For example, when a step modifies a component C1 and this component is linked to another component C2 by *Affects* link, the system is aware of the necessity of generating a substep to modify C2 to take into account the consequence of C1 change. This awareness is possible because of: first, the link that ties the two components and second, the existence of a rule specifying when to generate an automatic substep.

Ideas similar to the support functions discussed in this subsection were originally defined by Luqi in [47]. These definitions were enhanced and augmented in [69] to support the evolution process of the prototype system design as part of the ECS.

The lower level teamwork support provided in B.1 is the same as that provided by the ECS. Teamwork support within the context of the third rule is an extension of this support functionality. We introduce this extension that can support not only the concurrent work of individual designers, but also support the concurrent work of subteams. Each subteam includes analysts and designers who are assigned the responsibility of completing the analysis and resolving the issue as explained above.

Support related to scheduling and propagating inherited properties (see B.2 and B.3) are the same as that provided by the ECS. However, the scope of application is wider in our case. The scheduling process is also subject to more constraints especially in the analysis phase. The scheduling mechanism has to deal with the serialization of the analysis steps in the upper stream portion of the process as was explained in Chapter III. The serialization imposes another kind of precedence among analysis steps than the one explained in this section.

The structuring support we provide is an extension to that provided in [47]. This is because our model extends the one provided in the above reference. In our extended model we identify more software components (criticisms, issues, requirement and components). New type of steps is added (analysis steps). Individuals involved in the process are explicitly represented too. Also new types of relations among the model types are added.

C. DECISION SUPPORT COMPONENTS

Generally a decision support system (DSS) is a system that supports technological and managerial decision making by assisting in the organization of knowledge about ill-structured, semistructured or unstructured problems [3]. A decision support system primary goal is to increase the effectiveness of the decision-making effort which involves the formulation of alternatives, the analysis of their impact, and the selection of appropriate options for implementation. A decision support system should also have the ability to acquire, represent, and utilize information or knowledge.

A decision support system has the following main elements:

- Database component.
- Model component.
- User interface component.

The interaction between the decision maker and these components is shown in Figure 4.11 below [3].

1. The Database Component

The database contains the reservoir of information that describes all of the conditions and characteristics of the problem in question [58]. Through the employed database management system (DBMS), a better control over and management of data is achieved. Functions like, querying capability, sharing of data, backup and recovery, and enforcing of integrity constraints are examples of what is readily available of management and control functions provided in general by a DMBS.

DBMSs commercially available support different data models. A data model is a collection of data structures, operations that may be applied on the data structures, and a set of integrity rules to be enforced by the database system. Examples of these data models are: the relational, hierarchical, network, and object-oriented data models.

Our Decision support mechanism is supported by an object-oriented-based database. We selected the object-oriented data model for its suitability and representation capability for inherently complex and data intensive problems such as the one on hand. An object-oriented data model has the advantage of modeling all of the conceptual entities with a single concept, namely, objects [34]. It can easily represent a real world problem because:

1. It includes facilities to manage the software engineering process in general like data abstraction and inheritance.
2. It can easily represent complex objects.
3. It can directly represent relationships and interconnections in the data.
4. Properties of data are also directly available.

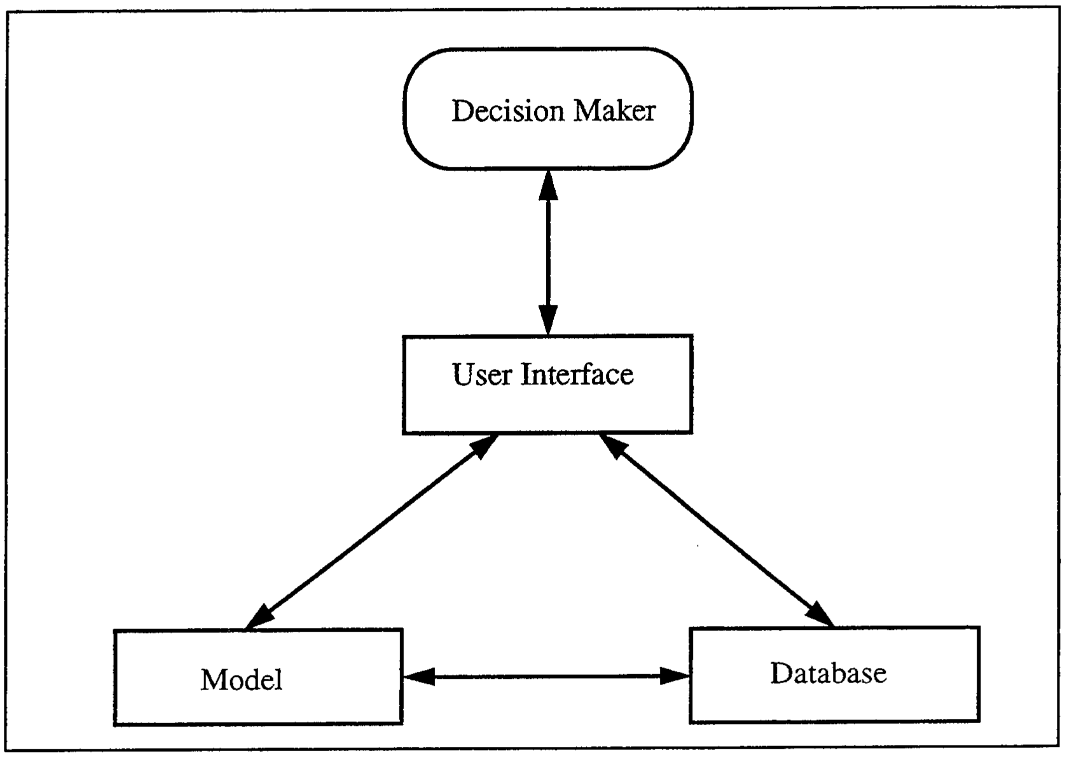


Figure 4.11 Decision Support System Main Components

In our view, the requirements elicitation, evolution, and the complex relations that tie this process entities to other parts of the system design and implementation can better be represented by objects, their properties, and their relationships to other objects. Within the context of an object-oriented database, it is also easier to track the history of an object along with its relationships which is very important in evolutionary process like ours. Recording and querying design rationale is another important outcome for design replay, learning process, comparative studies, and change in customer goals.

2. The Model Component

The modeling component of a DSS provides the means for mathematically representing the complex structure and relationship between the various parts of the problem elements. Modeling a real world problem is necessary in order that the problem can be depicted and analyzed by a computer. The mathematical formulation embedded within the model provides a descriptive mechanism through which information can be manipulated repeatedly and the decision maker can emulate what will actually take place [58].

In our problem, the model component has the following parts:

1. The conceptual model given by the extended graph model. This conceptual model provides the formalism of the process to:
   - Impose a well defined structure on the problem.
   - Represent the underlying relationships.
   - Control the evolution of the system as a whole.
   - Expose consequences of changes and suggest activities to make these changes consistent system-wide.
   - Assist in identifying issues to be resolved.
   - Explore alternatives available to resolve issues.
   - Assist in analyzing alternatives and selecting the most promising one according the measures specified by the decision maker.
2. A scheduling model which supports the scheduling of the analysis and design tasks under different timing, resource, and priority constraints as well as coping with the dynamics of the process.
3. Assignment model to draw from a common analysts and designers pool to be allocated to tasks under the constraints of tasks skill requirements, availability of designer, urgency of tasks, etc.
4. A set of rules and constraints that governs the process and is used in building algorithmic procedures that assist in automating many activities and performing different computations as explained in section A. These rules and constraints are also used to infer values and relationships and draw conclusions based on the given facts.

3. The User Interface Component

In a DSS, the user interface component is a prime motivation for the system that provides an effective interface with the user. One of the good attributes of a DSS is making available a user interface that provides all interactions between the computer and the decision maker and further hides the technical complexities and internal mechanism necessary to automate the process. Other desirable functions to be performed by a user interface are [11], [58]:

1. Translating user inputs into the appropriate directions for the computer.
2. Checking the validity and logic of all user inputs.
3. Generating appropriate and informative responses that explain the results, recommended possible creative actions, or suggest new direction to be evaluated.
4. Minimizing the user's memory load.
5. Speaking the user language.
6. Permitting easy reversal of actions.
7. Being consistent.

The user interface component of our system is a graphical user interface developed using Transportable Application Environment Plus (TAE Plus) [91]. Part of this user interface, let us call it the general part, represents parts of the CAPS user interface and is used by our support mechanism but is not specific to requirements analysis subsystems. The other part, to be called the specific part, is to be developed specifically for requirements analysis, evolution control, and interfacing to the design database.

(1) **The General User Interface:** This part represents the portion of the CAPS user interface relevant to the requirements process. It offers the following facilities [87]:

- Testing Display that shows the prototype execution or the interpretation of the design. The display may be a time chart indicating the system state change or the desired system behavior sequence like dialogue, input/output, reactions, etc.

- Analysis Display that shows the results of the static analysis as the designer required.

These two facilities are used in demonstrating the prototype to the customer to stimulate him to respond to the demonstration by criticisms. These criticisms are used after analysis for evolving the prototype to the next version within the process of firming up requirements.

(2) **The Specific User Interface**: This represents the part of user interface to be developed specifically to support the requirements process. It includes facilities to do the following:

- Map the process knowledge to the conceptual model objects,
- Establish relationships between the model objects.
- View and update values and dependencies previously stored in the database.
- Control analysis and design activities.
- Conduct on-line analysis of alternatives available for resolving issues.
- Support customers conduct the formal debate and judgement of available alternative requirements changes.
- View the consequences of selecting an alternative.
- Monitoring the execution of the plan and allow for dynamic changes in that plan.

This part supports the flow of information into and from the database and presents it in a user-friendly format. It includes varieties of facilities that allow its user communicate easily with the system like pull-down and pop-up menus, dialog boxes (e.g., radio buttons and check boxes), text entry fields, scrolable list of choices, etc.

V. SUPPORTING STAKEHOLDER DELIBERATION AND JUDGEMENT

In Chapter IV. we discussed how the decision support mechanism assists in generating alternatives to resolve open issues. We also discussed how the mechanism assists in gathering information to be used in the analysis of the generated alternatives in support of the resolution process. This information is presented to the stakeholders to assist them in the selection process. In this chapter we explain how can we choose an alternative or alternatives that reflect the customer preference.

A. A METHOD FOR CHOOSING AMONG ALTERNATIVES

The method we use is inspired by the Analytic Hierarchy Process (AHP) which underlies a mathematical model and process to prioritize options. The outcome priorities are then used to choose among alternatives. The method was successfully used in the areas of planning, resource allocation, conflict resolution, prediction, and other applications. Recently this technique was evaluated as more trustworthy and less time consuming than techniques like numeral assignments when used to prioritize requirements [38]. The material presented in this section related to the AHP model is based primarily on the many published works on the model by Thomas L. Saaty. Our focus is the application of the process part of the model is primarily to select among requirement alternatives that satisfy the customer needs, and be simple and cost effective, when implemented, as a secondary goal. For details concerning the mathematical model, validation, and different application of the AHP, refer to [77]-[80].

1. The AHP Process

Although the method is based on a mathematical model, the AHP process, its application, and use is very simple. For the sake of illustration, assume that we are to choose one (or more) from four alternatives *A1*, *A2*, *A3*, and *A4* to achieve the goal *g*. The process should end by answering the question: which of these alternatives is more important for achieving g. To solve this decision problem, the decision maker has to compare the alternatives pairwise, assigning each an importance factor relative to the other alternatives. This effectively, is filling an $n \times n$ importance matrix where n is the number of alternatives (n = 4 in our case). An entry $a_{i,j}$ in this matrix means that the importance of the alternative A_i relative to A_j is $a_{i,j}$ (with respect to the criteria under consideration). The matrix for the above example is shown in Figure 5.1. In reference to this figure, we can see the following:

1. Alternative *A1* is 5 times more important than *A2* (reflected by the matrix entry $a_{1,2}$), is 6 times more important than *A3* (reflected by the matrix entry $a_{1,3}$), and 7 times more important than *A4* (reflected by the matrix entry $a_{1,4}$).
2. Alternative *A2* is 4 times more important than A3 (reflected by the matrix entry $a_{2,3}$), and is 6 times more important than *A4* (reflected by the matrix entry $a_{2,4}$).
3. Alternative *A3* is 4 times more important than *A4* (reflected by the matrix entry $a_{3,4}$).
4. Six entries ($a_{1,2}$, $a_{1,3}$, $a_{1,4,}$ $a_{2,3}$, $a_{2,4}$, and $a_{3,4}$), out of the 16 total elements of the matrix, are the only entries that need to be filled in. The rest of the elements can be automatically determined according to the following rules.

a. The Main Diagonal Rule

It is intuitive that the importance ratio of an alternative relative to itself is 1. Therefore the main diagonal elements of the importance matrix *M* are filled in by 1's. Formally:

Rule1:

For all $a_{i,i}$ *such that* $a_{i,i} \in M,\ 1 \le i \le n \Rightarrow a_{i,j} = 1$, where n is the number of the available alternatives, and M is the importance matrix whose dimension is $n \times n$. This rule is used to automatically fill the main diagonal elements of M.

g	A1	A2	A3	A4
A1	1	5	6	7
A2	1/5	1	4	6
A3	1/6	1/4	1	4
A4	1/7	1/6	1/4	1

Figure 5.1 An Importance Matrix

b. Reciprocals Rule

In comparing a pair of alternatives A_i and A_j with respect to a criterion c, if A_i is judged as three times more important than A_j, then it is intuitive that A_j is one third the importance of A_i with respect to c. In terms of the importance matrix, this fact can be expressed formally by the following rule

Rule2

For all a_{ij} such that $(a_{i,j} \in M), a_{j,i} = 1/a_{i,j}$.

The reciprocals rule is used to automatically fill in the elements of the importance matrix that lie below the main diagonal. This rule along with the main diagonal rule reduces the number of the pairwise comparison from n^2 to $(n(n-1))/2$ comparisons.

2. The Pairwise Comparison Scale

To fill in the importance matrix as a result of the pairwise comparison process between alternatives, a scale is needed. The scale is used to represent the relative importance of one alternative over another with respect to the comparison criterion. The AHP model uses a numerical scale that has values from 1 to 9. A scale of 9 units is reasonable and reflects the degree to which human judgement can discriminate the intensity of importance between alternatives. The AHP scale used for pairwise comparison is given in Table 5.2.

Unit	Meaning
1	Equal importance of 2 alternatives.
3	Moderate importance of one over another.
5	Essential or strong importance.
7	Very strong importance.
9	Extreme importance.
2, 4, 6, 8	Intermediate values between the two adjacent judgement used when compromise is needed between two judgements.
Reciprocals	In comparing an alternative A_i with A_j, if A_i is assigned one of the above values, then A_j is assigned its reciprocal.
Rationals	Ratios arising from forcing consistency of judgement.

Table 5.2: The Pairwise Comparison Scale

B. ALTERNATIVES WEIGHTING METHODOLOGY

In the pairwise comparison process stakeholders assign relative importance to each alternative with respect to the others. The pairwise comparison is expressed using the importance matrix. The next step after filling in the importance matrix is to determine global weights for all the alternatives. The weighting function can be thought of as defining priorities of alternatives. The higher the priority of an alternative, the more likely the goal is achieved by implementing that alternative than any other alternative that has a lower priority.

The global priorities of the alternatives are obtained by manipulating the importance matrix resulting from the pairwise comparisons. The result of this manipulation is a vector of priorities that has an entry for each alternative as we explain below.

1. Priorities Vector

Given the importance matrix M, the priority vector is given by the *normalized principal eigenvector of M*. The principal eigenvector is the eigenvector corresponding to the largest eigenvalue λ_{max}. The mathematical justification for this result is given in section B.3 below. The desired eigenvector is given by the normalized row sum of the limiting power of the importance matrix. This is obtained by raising the matrix into some arbitrary large power and dividing the sum of each row by the sum of the elements of the matrix. Since the cost of this kind of computation is expensive in terms of time, some other methods are used that give crude estimates of the required vector. One of these method called *averaging over the normalized columns* is summarized in the following steps:

1. Sum each column in the importance matrix M.
2. Divide each element in M by the sum of the column it belongs to obtaining $M1$.
3. Sum each row in $M1$, the result is a vector v.
4. Divide v by the sample size of n columns, the result is the required priority vector.

Applying these steps to the importance matrix of the example given in Figure 5.1, the corresponding matrix $M1$ and the resultant priority vector are shown in Figure 5.2. As can be seen from this figure, the alternative $A1$ is the most promising alternative to achieve the goal g with intensity of importance 61 percent. If it happens that two alternatives are very close in their importance values, then both can be chosen. Prototyping both and comparing their effects, determines the final judgement as to which is better.

M1

g	A1	A2	A3	A4	Priority Vector
A1	0.66	0.78	0.53	0.93	0.62
A2	0.13	0.16	0.36	0.33	0.24
A3	0.11	0.04	0.09	0.22	0.10
A4	0.09	0.03	0.02	0.06	0.04

Figure 5.2 The Matrix M1 and the Priority Vector

2. Consistency in Selecting Alternatives

In comparing alternatives pairwise, if alternative *A1* is judged as 3 times more important than *A2*, and that *A2* is judged as 2 times more important than *A3*, then for the judgement to be consistent, *A1* should be judged as *3x2 = 6* times more important than *A3*. In terms of the importance matrix entries this is expressed as $a_{1,3} = a_{1,2} \times a_{2,3}$. In general for an importance matrix to be consistent, the following condition must hold:

$a_{i,k} = a_{i,j} \times a_{j,k}$ *for all* $i,j,k = 1,.., n.$

This represents an ideal case which is not normally true in practice. If this condition is violated for some entries, then a judgement error occurs. The AHP model accounts for such situations and provides for the measurement of inconsistency if it occurs. Deviation from consistency can be represented by the consistency index CI:

$$CI = (\lambda_{max} - n)/(n-1) \ .$$

Where λ_{max} is the maximum or principal eigenvalue of M and n is the number of alternatives being evaluated. According to the AHP theoretical results, λ_{max} has the following properties:

1. $\lambda_{max} > 0$.
2. $\lambda_{max} \geq n$.

An approximation of λ_{max} can be obtained by the following procedure:

1. Multiply the importance matrix on the right by the priority v vector to obtain a new vector $v1$.
2. Divide each element of $v1$ by the corresponding element of v to obtain a new vector $v2$.
3. Sum the elements of $v2$ to obtain the sum S.
4. Divide S by the number of elements to obtain an approximate value for λ_{max}.

To measure the consistency of an importance matrix M, the CI of M is compared to a randomly generated consistency index (Random Index) of a matrix of the same order and conforms to the reciprocals rule. Random Indices (*RIs*) for matrices of the order from 1 to 15 were generated and tabulated [77]. Each *RI* is the average of *RIs* of the same order matrices using a sample size of 100. Table 5.3 gives the matrix size and its corresponding average RI for matrices of sizes 1 to 7.

Matrix Size	Average RI
1	0.00
2	0.00
3	0.58
4	0.90
5	1.12
6	1.24
7	1.32

Table 5.3: Average RIs for Randomly generated Importance Matrices

The ratio of *CI* to the average *RI* of the same order matrix, called *Consistency Ratio* (*CR*), is used to measure the consistency of an importance matrix. Through experience, a ratio of 0.1 or less is considered acceptable. If a pairwise comparison results in an unacceptable *CR* by a high margin that can not be tolerated, then another new round of pairwise comparisons is performed to bring the *CR* to a tolerable value.

For our running example, $\lambda_{max} = 4.39$. Using this value with n = 4 we get the consistency index $CI = (4.39 - 4)/ (4-1) = 0.13$. The corresponding average RI for a matrix of this size (see Table 5.3) is 0.9 which when used with the computed CI gives a consistency ratio $CR = (0.13/0.9) = 0.14$ which is slightly higher than the threshold value (0.1).

3. Mathematical Justification

The basic mathematical justification of the method given below was first introduced in [80]. This justification answers the question: why is the priority vector given by the principal eigenvector of the importance matrix, and what does the corresponding eigenvalue (λ_{max}) have to do with the consistency of judgement?

Assume that we have an *nxn* importance matrix *A* with elements $a_{i,j}$. This matrix represents a pairwise comparison of *n* activities (e.g., alternatives) with respect to some criterion of judgement. We want to find a global weights of influence $w_1, w_2,.., w_n$ of all activities on the criterion. We know from above that *for all i,j =1,2,.., n*:

1. $a_{i,j} > 0$
2. $a_{i,j} = 1/a_{j,i}$.
3. $a_{i,i} = 1$.

Ideally, if judgement is perfect, e.g., based on known exact measurements, then the global weights $w_1, w_2,.., w_n$ are known. Further *A* is consistent, i.e in addition to the above three characteristics, *A* satisfies:

$a_{i,k} = a_{i,j} \times a_{j,k}$ *for all i,j,k = 1,.., n.*

With w_i, *i, =1,2,.., n,* known, the following holds for the matrix A *for all i,,j,k =1,2,.., n*:

$$a_{i,j} = w_i/w_j.$$

$$a_{i,j}a_{j,k} = (w_i/w_j) \times (w_j/w_k) = ((w_i)/w_k) = a_{i,k}$$

which proves the consistency condition in the ideal case. We also have:

$$\left(a_{i,j} = \frac{1}{w_j/w_i}\right) \Rightarrow \left(a_{i,j} = \frac{1}{a_{j,i}}\right) \Rightarrow \left(a_{i,j} \cdot \frac{w_j}{w_i} = 1\right)$$

Using the above results and by direct substitution the following holds:

$\sum_{j=1}^{n} a_{i,j}\frac{w_j}{w_i} = \sum_{j=1}^{n} (a_{i,j} \cdot w_j/w_i) = n$ or equivalently:

$$\sum_{j=1}^{n} a_{i,j} \cdot w_j = nw_i \text{ which is equivalent to: } \boxed{\mathtt{Aw = nw}}$$

This is the well known eigenvalue problem, where n is the eigenvalue of A and w is the corresponding eigenvector.

In practice $a_{i,j}$ is not based on exact measurement as in the ideal case, but on subjective judgements. Therefore $a_{i,j}$ deviates from the ideal ratio (w_i/w_j) and the last equation no longer holds. However the following two facts from linear algebra help in this situation:

1. For a matrix $A(nxn)$, $\sum_{i=1}^{n} \lambda_i = trace(A)$, where λ_i, $i = 1,2,.., n$ is an eigenvalue of A, and *trace(A)* is the sum of the n main diagonal elements of A [29]. In our ideal case, we have $a_{i,j} = 1$ *for all* $i = j = 1, 2,.., n$. Hence $\sum_{i=1}^{n} \lambda_i = n$ i.e all eigenvalues are zero except one which is n.
2. For a matrix $A(nxn)$, if A is ***positive*** (i.e $a_{i,j} > 0$, not to be confused with *positive definite*) and ***reciprocal*** (i.e $a_{i,j} = 1/a_{j,i}$) *for all* $i,j = 1, 2,.., n$, then if we change $a_{i,j}$ by small amounts, the eigenvalues of A change by small amounts too. Hence in the practical case, the eigenvalues of the judgement matrix remain close to those of the ideal case: the largest eigenvalue λ_{max} is close to n and the others are close to zero. Hence the priority vector of a judgement matrix whose entries deviate from the ideal consistent one by small amounts is given by the priority vector corresponding to the maximum eigenvalue of that matrix.We conclude that given the pairwise comparison matrix A, we can find the priority vector w: we solve the following equation for w: $\boxed{Aw = \lambda_{max}w}$.

To have a unique solution and makes the sum of priorities equal unity, we normalize the solution by dividing each element in w by the sum of the elements.

In the consistent (ideal) case, we have n as the largest eigenvalue. If a_{ij} deviates by small changes, the resulting largest eigenvalue is λ_{max} which is always larger than n. The more consistent the judgement matrix is the less the difference between both. In the ideal case both are equal. Therefore the deviation of λ_{max} from n is a measure of consistency. That is why the value $\frac{\lambda_{max} - n}{n - 1}$ is taken as a consistency index to indicate closeness to consistency.

C. MULTI-LEVEL HIERARCHY DECISION PROBLEMS

Most decision problems are semi or ill-structured. A necessary prerequisite for providing efficient solutions for such problems is structuring. Hierarchical structuring is a structuring technique that well suits decision problems in general. A hierarchy abstracts a system structure in a way that exposes the system components. Hierarchical structure is also close to human thinking.

The AHP employs hierarchical structuring as the underlying abstraction technique. A problem is broken down into a number of hierarchical levels. The number of levels and what each level represents depends on the nature of the problem and the sought solution. In our case, the problem is to choose from alternatives based on some criteria. Therefore the problem structure in general includes three main levels:

1. **The Focus**: is the root of the hierarchy (level1).
2. **The Criteria**: are the attributes on which judgement is based (level 2).
3. **The Alternatives:** to choose from (level3).

Levels 2 and 3 can be further decomposed into more levels to detail criteria to subcriteria and to qualify alternatives to be more concrete. Regardless of the number of levels in the problem structure, the application of the method remains conceptually the same. In the above subsection we discussed how, given the importance matrix, we can compute the priority vectors and consistency ratios. This discussion was applicable to only one specific level of the problem hierarchy with respect to one criteria in a lower level. In the following we extend these ideas to be applicable to the hierarchy as a whole [80].

1. Composite Priority Vector

Informally, in a hierarchy H with h levels, if the priority vector of the pth level with respect to some element z in the (p-1)st level is given by the vector V_p, then the priority vector of the qth level with respect to z where the qth level is higher than the pth level (the root is lowest level in the hierarchy), is given by [79]:

$$W = B_q B_{q-1} \ldots B_{p+1} V_p$$

Where B_k is the priority matrix of the kth level.

Therefore the priority vector of the highest level with respect to the root element (the root of the hierarchy; to be termed *Focus* in the next subsection) is given by:

$$W = B_h B_{h-1} \ldots B_2 V_1$$

Where V_1 is the priority vector of the second level with respect to the root elements (a scalar taken as 1 if the hierarchy has only one element in the lowest level as normally is the case).

In the case of three level hierarchy of a single element in the first level, the last equation reduces to multiplying the third level priority matrix on the right by the priority vector of the second level. The intuition of the general rule is clear in this case: this is effectively the same as weighting each eigen vector in the third level by the priority of each element in the second level and then adding. In the case of choosing between alternatives, the intuition is even clearer. To get an overall ranking of each alternative, we need to multiply the weight indicating the qualification of that alternative with respect to a specific criterion by the weight of that criterion in the selection process.

2. Composite Inconsistency

The measurement of consistency in judgement can also be generalized to the entire hierarchy. The intuitive approach would be multiplying the index of consistency obtained from a pairwise comparison matrix by the priority of the element (or criteria) with respect to which the comparison is made and then add all the results for the entire hierarchy. This intuition is formalized below [80].

Let n_j, j = $1,2,\ldots, h$ be the number of elements of the jth level of a hierarchy of h levels. Let w_{ij} be the composite priority of the ith element in the jth level, and let $k_{i,j+1}$ be the consistency index of all elements in the $(j+1)$st level compared with respect to the element i of the jth level. The consistency index of the hierarchy is given by:

$$C_H = \sum_{j=1}^{h} \sum_{i=1}^{n_{i_j}} w_{ij} k_{i,\,j+1}$$

Where $w_{ij} = 1$ for j = 1, and n_{i_j} is the number of elements in the jth level with respect to which the elements of the $(j+1)$st level are compared.

The result of the above sum is then compared with the corresponding index obtained by taking randomly generated indices, weighting them by the priorities, and adding.

D. IMPROVEMENT OF THE AHP

We propose the following improvements on the AHP method. These improvements are concerned with:

1. Improving the AHP scale.

2. Improving the efficiency of computation of the method.
3. Improving the accuracy of the results.
4. Changing the way the method is applied to allow stakeholders express their independent judgements.
5. Combining our view of the IBIS model with the AHP to overcome the limitations of both the IBIS model and the AHP.

The first improvement makes strictly consistent judgements come out in more natural way. The second and the third improvements involve the use of exact (or very close to exact) methods to compute the priority vectors and consistency indices. The fourth improvement changes the way the method is applied for problem solving. This improvement synthesizes the solution of the decision problem from the solutions of multiple instances of the same problem and then combines them. In the requirements context this improvement has the spirit of combining parallel elaborations [59]. The fifth improvement is to combine our version of the IBIS-based process with the AHP process. This superimposition is used in formalizing the notion of the IBIS position as we explain in section C.3 and quantify judgements. The same improvement adds some representation capability to the AHP using the IBIS model objects and relationships.

1. Improving the AHP Scale

The problem with the original weighting scheme is that for the official weightings there often do not exist choices for some of the entries that would make all the choices consistent. Consistent judgement requires transitivity. The condition of transitivity in terms of the comparison matrix entries is expressed by the following equation which we discussed earlier in this chapter and is repeated here:

$a_{i,j} \times a_{j,k} = a_{i,k}$ *for all i,j,k = 1,.., n*

Consider three alternatives A, B, and C. According to the AHP levels of importance if A has strong importance (corresponds to the scale value 5) over B, and if B has moderate importance (corresponds to the scale value 3)over C, then for the consistency to hold the importance of A with respect to C must be $strong \times moderate = 5 \times 3 = 15$ which is outside the numerical range (1-9) of the AHP scale.

Therefore as an improvement we suggest the numerical weightings of the importance levels be distributed exponentially over the scale levels. This makes the scale more dense and to some extent closed under multiplication.

Table 5.4 shows a fragment of an example exponentially distributed scale. The entries of this scale are given by the geometric series: b^0, b^1, b^2,.. , b^8. According to this new scale $strong \times moderate = b^5 \times b^3 = b^8$ which lies within the scale

IMPORTANCE LEVEL	THE AHP SCALE	THE EXP. SCALE	VALUES
Equal Importance	1	b^0	1
	2	b^1	1.3
Moderate Importance	3	b^2	1.7
	4	b^3	2.3
Strong Importance	5	b^4	3.0

Table 5.4: The Exponential Scale

For simplicity of use the entries of the new scale can be mapped to the original scale. Since we know that b8 = 9, we conclude that b = 1.3 from which the rest of entries can be found as shown in the table. The user can then be presented with the AHP scale and the system maps it to the exponentially distributed one.

2. Computation Improvement

According to the mathematical model of the AHP [78], the following theorem is used to compute the eigenvector corresponding to the maximum eigenvalue:

$$\lim_{k \to \infty} \frac{A^k e}{e^T A^k e} = c w_{max}, \text{ where:}$$

A is the importance matrix, c is a constant, $w_{\max}$ is the eigenvector corresponding to the maximum eigenvalue, and $e = (1, 1, \ldots, 1)^T$. Informally it states that the principal eigenvector ($w_{\max}$) is given by the normalized row sums of the limiting power of the importance matrix. So a suggested way to calculate $w_{\max}$ is to raise A to powers that are successively squared each time. The row sums are calculated and normalized. As a stopping rule, the computation ends when the difference between these sums in any two consecutive iterations is smaller than a predefined small value.

The complexity in terms of time of the above outlined algorithm is $O(n^3 \log_2 k)$ which is expensive computation cost that let the AHP use approximate methods as was explained in B.1, see page 97. The use of these methods produces crude estimates of the desired values [80].

However there are known algorithms that give more accurate results and they are more efficient in the same time. Many of these algorithms are based on and augmentation of the well known *Q-R* algorithm from matrix theory. Some variants of the *Q-R*-based algorithms solve the eigenvalue problem in $O(n^2)$ [29]. This is a great improvement over the complex limiting power algorithm of the AHP. Even it is comparable to the complexity of the crude estimate algorithm the AHP uses. Moreover the *Q-R-based* algorithms should give results more accurate than both the limiting power and the crude estimate algorithms. The details of the *Q-R-based* algorithms for solving the eigenvalue problem is beyond the scope of our work. For more details refer e.g., to [27].

Therefore our first improvement to the AHP is to use any of the available Q-R-based algorithms to find the priority vectors and λ_{max}. However many linear algebra packages are available today that solve the same problem efficiently using the Q-R-based algorithms. An example is MATLAB [92]. The only problem with MATLAB is that the results are not normalized. So we need to first, find the maximum eigenvalue and the corresponding eigenvector, and then divide the elements of the resulting vector by the sum of the elements of the vector. This normalizes the elements and makes their sum equal unity. These steps are accomplished by feeding the MATLAB results into a program we wrote (Prog1) to do the required postprocessing.

We also wrote a second MATLAB program (Prog2) that computes the priority vectors for matrices of different dimensions using the AHP crude estimates algorithm. We ran the two programs on sample square matrices of dimensions from $n = 4$ *to* $n = 9$ to compute the priority vectors. The MATLAB computation is comparable in efficiency to but more accurate than the AHP crude method. A fragment of these results is shown in Figure 5.3. The figure gives the comparison matrix (7x7), Priority Vector1 as computed by Prog1 and Priority Vector2 as computed by the crude algorithm. Appendix C includes the source MATLAB code for both programs and the results of running both on different matrices as well as the resulting priority vectors.

```
                             Comparison Matrix

  1.0000    4.0000    9.0000    6.0000    6.0000    5.0000    5.0000
  0.2500    1.0000    7.0000    5.0000    5.0000    3.0000    4.0000
  0.1100    0.1400    1.0000    0.2000    0.2000    0.1400    0.2000
  0.1700    0.2000    5.0000    1.0000    1.0000    0.3300    0.3300
  0.1700    0.2000    5.0000    1.0000    1.0000    0.3300    0.3300
  0.2000    0.3300    7.0000    3.0000    3.0000    1.0000    2.0000
  0.2000    0.2500    5.0000    3.0000    3.0000    0.5000    1.0000

                              Priority Vector1
  0.4273    0.2304    0.0206    0.0524    0.0524    0.1226    0.0943

                              Priority Vector2
  0.4084    0.2264    0.0215    0.0577    0.0577    0.1277    0.1002
```

Figure 5.3 Priority Vector Computed by the Two Methods

3. Combining Parallel Judgements

In the original AHP, a decision problem is represented by a single instance of a hierarchy structure. The elements in one level of the hierarchy (e.g., alternatives) have only one importance matrix with respect to each one of the elements in the next lower level (the root is at the lowest level). This next lower level may represent the criteria used for judging alternatives. If a group of decision makers are involved in solving such a problem with this structure, they have to agree on the importance matrices of all elements in all levels. Beside being a tedious work to do, this does not seem appropriate in judgement situations where different individuals with different opinions are involved in the process. What we need is a structure that allows every stakeholder to explicitly and independently express his view point and then combine these view points in some way. The final outcome is a group decision while keeping each individual's judgement intact for future references. This is the theme of our next improvement which we elaborate below.

Our solution to the above problem is to use multiple instances of the same decision problem. Each instance is used by one of the participating stakeholders. Each stakeholder uses the instance to establish the pairwise comparisons in all levels of the instance hierarchy according to his preference. The system uses these comparisons to carry out the intermediate computation in all levels of the hierarchy. The final outcome is a vector of priority that weighs the available alternatives and a consistency index. Both reflect that person's individual judgement.

Once the individual judgements are complete by all stakeholders, they are fed to another manipulating process that combines them and comes out with a group judgement. This latter result reflects the combined preference of all the stakeholders. The computation here is automatic and does not require the intervention of the stakeholders. However the manipulating process does need a uniform ranking list of all the stakeholders participating.

This can be done by an *executive board* which assigns a rank (from the AHP scale) to each stakeholder. The *executive board* members assign equal ranks to themselves. This ranking list is used by the system to automatically establish importance weights for all the stakeholders. These weights represent the priority vector of the stakeholders and are derived from the ranking list by simple manipulation of this list as follows:

$$w_i = \frac{r_i}{\left(\sum_{i=1}^{n} r_i\right)}$$ where w_i and r_i are the weight and the rank of the stakeholder i

respectively, and n is the number of the participating stakeholders. The result of this simple manipulation is the same as computing it using an importance matrix. This is possible in this special case because mapping the ranking list into an importance matrix makes this matrix consistent. We provide the following theorem to prove that in this special case the priority vector, though it is the result of a very simple manipulation, gives the same result as if we compute it using the ranking list and the AHP importance matrix.

Theorem: *The priority vector obtained from the ranking list by normalizing that list by the sum of its element is an eigenvector of the importance matrix constructed using the ranking list. Moreover this vector is a principal eigenvector of that matrix.*

Proof: Assume that the AHP is used to fill an importance matrix S using the ranking list for n stakeholders. An element $a_{i,j}$ in S is given by: $a_{i,j} = (r_i/r_j)$ where r_i and r_j are the ranks assigned to stake holder i and j respectively. Therefore:

1. Since ranks in the list are drawn from the AHP scale, then for all $a_{i,j}$, $i,j = 1, 2, .., n$, $a_{i,j} > 0$ which implies that S is positive (not to be confused with positive definite).

2. $a_{i,j} \cdot a_{j,k} = (r_i/r_j) \cdot (r_j/r_k) = (r_i/r_k) = a_{i,k}$ which satisfies the consistency condition discussed in B.2

3. Also we have $a_{j,i} = r_j/r_i = 1/(r_i/r_j) = 1/a_{i,j}$ *for all* $i,j = 1, 2, .., n$ which

means that S is reciprocal.

Now consider multiplying S on the right by our vector w:

$$Sw = \begin{bmatrix} \frac{r_1}{r_1} & \frac{r_1}{r_2} & \cdots & \frac{r_1}{r_n} \\ \frac{r_2}{r_1} & \frac{r_2}{r_2} & \cdots & \frac{r_2}{r_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{r_n}{r_1} & \frac{r_n}{r_2} & \cdots & \frac{r_n}{r_n} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_n \end{bmatrix} = \begin{bmatrix} \frac{r_1}{r_1} & \frac{r_1}{r_2} & \cdots & \frac{r_1}{r_n} \\ \frac{r_2}{r_1} & \frac{r_2}{r_2} & \cdots & \frac{r_2}{r_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{r_n}{r_1} & \frac{r_n}{r_2} & \cdots & \frac{r_n}{r_n} \end{bmatrix} \cdot \frac{1}{\sum_{i=1}^{n} r_i} \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_n \end{bmatrix} = nw$$

We started with Sw and reached to $Sw = nw$. This proves that w is an eigenvector of S whose corresponding eigenvalue is n. Satty in [80] proved that a consistent matrix satisfying the three conditions stated in B.3 has a maximum eigenvalue n. S satisfies these three condition as we have just shown in 1. through 3. above. Therefore w is a principal eigenvector (i.e corresponds to the maximum eigenvalue). This completes the proof.

Table 5.5 provides an example of a uniform ranking list for 5 stakeholders and the weights computed automatically by the system according to the simple manipulation we developed above.

Stakeholder$_i$	r_i	w_i
SH1	5	0.33
SH2	4	0.27
SH3	1	0.07
SH4	3	0.20
SH5	2	0.13

Table 5.5: Stakeholders' Weights

In this establishment the automation facility emulates the process of pairwise comparing stakeholders without actually doing it. What makes this possible is the total order imposed by the ranking list. This order enforces consistency in judging the relative importance among stakeholders. We enforce consistency in this case only to avoid the inappropriate way of pairwise comparing stakeholders.

The process of combining individual judgements into one group decision is summarized in the following steps and illustrated in Figure 5.4:

1. Compute the stakeholders priority vector as explained above.
2. Stakeholders decide the criteria to be used in judgement and the number of hierarchy levels used to break down these criteria.
3. The system uses this information along with the automatically generated information about the available alternatives to construct the problem hierarchy.
4. Every stakeholder participating is provided with an instance of the problem as constructed by the system.
5. Every participating stakeholder conducts the required comparisons in all levels independently according to his preference.
6. Using these comparisons, the system computes a preference vector for each stakeholder (w_{st}) that reflects his opinion. An individual's consistency index is also computed.
7. Once the computations related to all the individuals are complete, the system combines the results obtained in 6. into one group priority vector that reflects the combined group judgement. This latter vector is computed as follows:
    - Construct a combined priority matrix $M_c(lxn)$ where l is the number of alternatives and n is the number of participating stakeholders. M_c is filled column-wise. Each column in M_c is filled by the corresponding w_{st}.
    - Multiply M_c on the right by the priority vector (w) which gives relative

importance to stakeholders.

- The output of the above multiplication is a combined priority vector (w_g)that reflects the group decision in prioritizing the available alternatives given the agreed upon judgement criteria.

In summery our improvement has the following advantages over the original application of the method:

1. Provides each stakeholder with one instance of the decision problem to allow independent judgements which is more appropriate in our context.
2. Simplifies the decision problem structure and handling.
3. Represents disagreements.
4. Supports distributed decision making. The original application is more suitable to round the table kind of decision making.
5. Avoids the inappropriate way of comparing stakeholders.

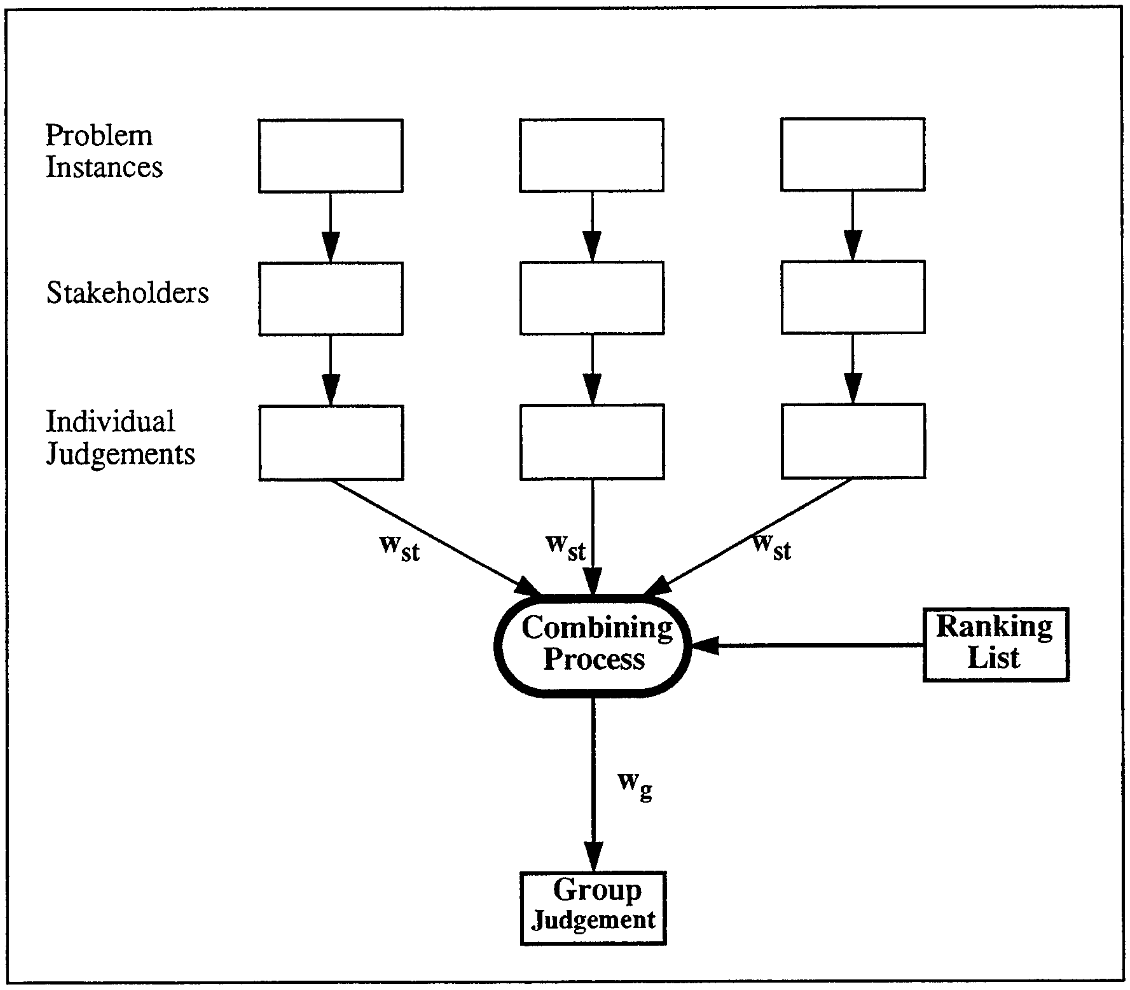


Figure 5.4 Combining Individuals Judgements into a group Decision

4. Combining IBIS and the AHP

With our improvements introduced above, the AHP works very well as a computational process. The method's strongest point is in providing judgements some kind of concrete quantification. However it lacks the representation of the rationale and justification for decisions. Also it is not capable of expressing the informal and rhetorical information inherent in the early design deliberation process. Therefore our next improvement is to combine the AHP and our view of the IBIS model. This combination extends the capability of the IBIS model and the representation power of the AHP.

Our version of the IBIS as was introduced in Chapter III. has, among others, the following differences from the original IBIS model:

1. Unlike IBIS, in our model issues have context and derivation way: they are synthesized from the criticisms posed by the stakeholders in response to the system demonstration.
2. Alternatives for resolving issues are identified using the requirements subsets affected by the issues. Hence the deliberation is more specific and is not drifted far from the subject debate. In the case of IBIS alternatives (positions) are based mostly on brain storming ideas in which deliberation can easily be drifted out of the context.
3. As a result an IBIS position in our context is represented by an object for the stake holder and his judgement in choosing among the available alternatives. We call this an individual's judgement profile or profile in short. Moreover, as we explained above, we combine these individuals' judgements into one group judgement profile or group profile for short. IBIS does not have the notion of group judgement (or group position).

The above improvements, while keeping the spirit of the original IBIS, provide better representation and control over the deliberation process. We are still keeping the strong points of the original IBIS. One of these strong points is that despite the fact that the IBIS model is based on some formality, yet it allows the recording of design rationale in a largely informal manner. This makes the design deliberation more flexible. However judgements in IBIS are not formal enough and moreover lacks quantification. In the IBIS model, judgement is based on an implicit binary scale. A stakeholder supports or objects a position. Also it is not clear in IBIS how decision are taken. Our understanding is that after the formal deliberation is complete, another informal deliberation is needed to combine the judgement of the deliberating individuals. This is needed to reach a decision by approving one or combination of the positions. So we suggest combining IBIS with the AHP motivated by the following:

1. IBIS serves in the representation of the design rationale along with their justification.
2. IBIS provides the types and relationships capable of representing the informal knowledge of the process.
3. The AHP provides the methodology for quantifying stakeholders judgements.
4. The AHP assists the task of combining judgements from different individuals to reach final decisions.

E. Q-IBIS

We call the model resulting from combining the AHP and the IBIS models with the above motivations, the Quantified IBIS or Q-IBIS for short. It has the IBIS types and relationships plus the following extensions and improvements. These extensions and improvements come form two sources: the first is the use of the AHP to quantify the IBIS positions. The second comes from our augmenting the IBIS model with more types to alleviate the following drawbacks in IBIS:

1. No explicit representation for stakeholders participating in the deliberation process.
2. No explicit representation for group positions.
3. No explicit representation for final decisions taken by the deliberating group.
4. No explicit representation for criteria of judgement. The existence of these serve two folds:
    - Make judgements more objective.
    - Assist in creating more accurate judgements. An issue looked at from different angles receives better judgement. Moreover, broken down criteria assist in synthesizing judgement from fine granularities.
5. No explicit representation for alternatives available for resolving issues. In require-

ments engineering, at least some kind of rough ideas about available alternatives is normally there. It is better to represent alternatives and incrementally refine them as more information become available than not representing them at all. In our context, availability of alternatives is more concrete where deliberation is performed in response to the demonstration of an executing prototype.

In the Q-IBIS we model the decision making problem as a network of types and relationships. Types are represented by the nodes. The model has seven types as shown in Figure 5.5. Relationships between types are represented by the links between nodes. The Q-IBIS types and relationships are shown in Figure 5.5 and explained below:

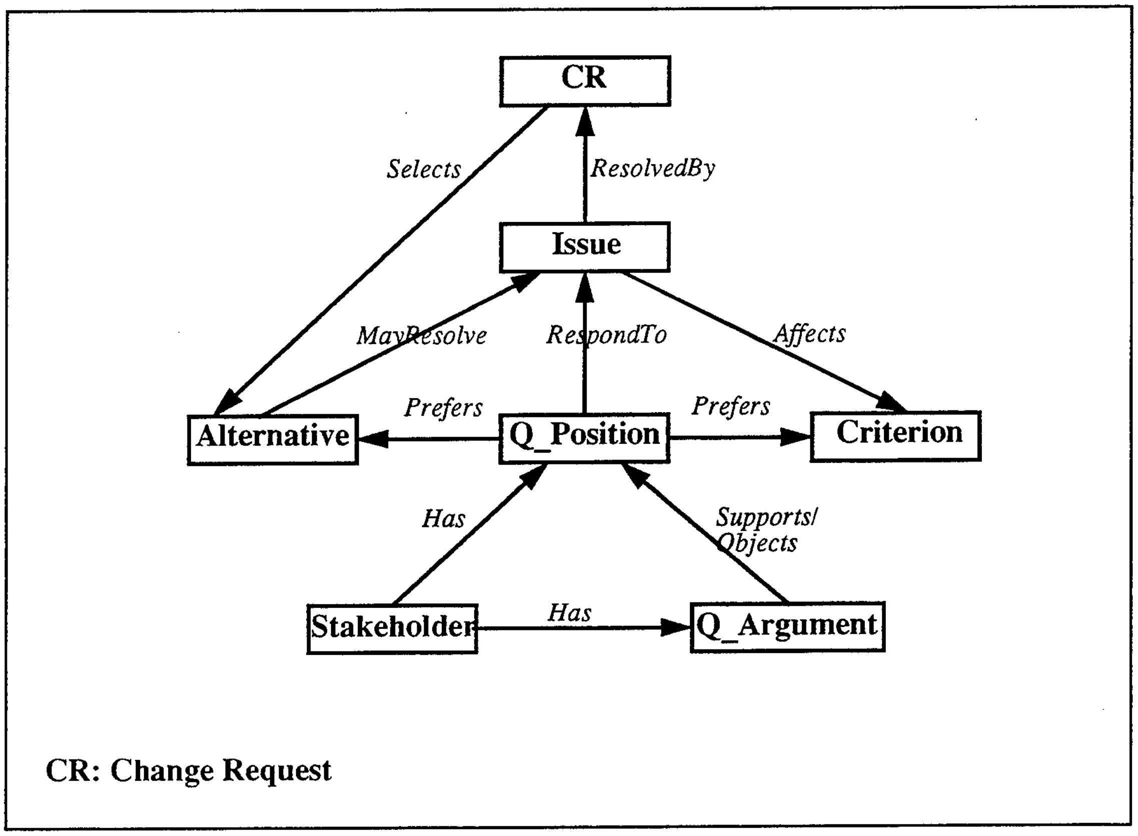


Figure 5.5 Q_IBIS Data Model

1. Types

1. An issues has the same meaning as in the original IBIS except that in our context

the set of issues are predetermined.

2. Q_position has the same general meaning as in the original IBIS but it differs from in the following:
    - Q_positions are quantified judgement based on a formal model of quantification.
    - Q_positions are further specialized into two subtypes: QI_position and QG_position, for quantified individual and group positions respectively.
    - QI_position represents the preference of a specific stakeholder in resolving an issue using the available alternatives. It has attributes that carry an individual judgement profile.These include the pairwise comparison matrices as constructed by him, different importance (for criteria) and preference (for alternatives) vectors, and different consistency indices.
    - QG_position represents the combined preference of all the participating stakeholders. It has similar attributes to the above but it carries the group judgement profile.
3. Q_argument has the same general meaning as an argument in the original IBIS but here it is, like Q_position, further specialized into QI_argument and QG_argument with the same motivation. This type has textual attributes that describe the reasons or any assumption behind an individual or the group preference.
4. Criterion of judgement to guide stakeholders in the selection process and make judgements more subjective. Stakeholders must agree from the beginning on the set of criteria and their detail. The detail in the criteria hierarchy is represented by PartOf relationship.
5. Alternative includes the requirement subset to be manipulated for resolving an issue. It also includes the kind of manipulation required. The manipulation is concerned with changing, deleting, or creating new requirement components.
6. Change Request (CR) represents the group final decision to resolve an issue. It specifies the selected alternative or alternatives and what the exact requirements change needed in the requirements subset of each selected alternative.This type

allows the stakeholders to also override the automatic process decision if the group sees a reason for. Many attributes can be valuable here. Deadline for accomplishing the work required by the change request, and a textual description that record the statement of the new requirements or the changes in existing ones are examples.

2. Relationships

Our extension and improvement of IBIS introduces new relationships. Some of these relationships has no correspondence in IBIS while others are improvements over IBIS ones. The improvement makes these relationships more expressive and objective. Some of the original IBIS relationships are not needed in our context. For example the generalizes/ specializes relationships that links an issue object to another is not needed during the deliberation process. Deliberation is conducted on a predetermined issues and alternatives. Issues are synthesized and analyzed before being deliberated. However our Q-IBIS is flexible enough to include such relationships if needed in another context.

1. *Has*: links a stakeholder to Q_position or Q_argument(s). It is one-to-one when it links a stakeholder and QI_position and many-to-one in the case of QG_position. It is one-to-many in the case if Q_argument is specialized to QI_argument and many-to-many if it is specialized to QG_position.
2. *Supports/Objects*: links a Q_argument to a Q_position. These are original IBIS relationships. The value *supports* is the default value between a Q_argument and Q_position objects owned by the same stakeholder. Notice that a stakeholder has only one Q_position and maybe many Q_arguments. This allows a stakeholder to informally record his support or objection on others' Q_positions. Therefore this relationship is many-to-one in both specialization of the Q_argument.
3. *Prefers*: links a Q_position to a criterion or an alternatives. This relationship has an attribute *strength* that quantify the stakeholder preference. The value of this attribute is filled automatically from the priority vector that expresses that stakeholder preference for criteria or alternatives. This relationship is many-to-many in both cases

4. *Affects*: links an issue to the criteria affected by the issue. This relationship although apparently is one-to-many (one issue has many criteria), it is made many-to-many to reuse criteria among different issues.

5. *RespondsTo*: links a Q_position to an issue. A position responds to only one issue and an issue is responded to by many Q_positions. This means that the relationship is one-to-many.

6. *MayResolve*: links an alternative to an issue. This relationship is many-to-one.

7. *ResolvedBy*: links an issue to the change request that resolves it. Normally this is a one-to-one relation. However it is made one-to-many to account for situations where more than one alternatives are selected for resolving an issue. This selection has the purpose of comparative study of alternatives by prototyping more than one alternative.

8. *Selects*: links a change request to the alternative(s) selected to resolve an issue. This is many-to-many relationship for the same above reason.

F. THE APPLICATION OF THE IMPROVED AHP PROCESS

The application of the AHP process in our context fits in the alternatives evaluation and selection process. It supports stakeholders to express their individual judgements and assists in combining these into group decisions. By using the improved AHP, Q-positions are no longer based on a binary scale of judgement as in the original IBIS. Using Q-IBIS, a stakeholder can express his preferences for available alternatives in a more smooth and quantified measures. Also the use of the AHP enables the use of the criteria of judgement. This makes the judgement more objective and bases it on deeper considerations. The application of the AHP in this way makes direct involvement of the stake holders not only through criticizing the demonstrated system, but also sharing in the way these criticisms should be remedied.

The decision support mechanism assists stakeholders in some other way. It augments available alternatives with important information for the stakeholders to consider in their judgement. The available alternatives are attributed by this information. This information is computed for each alternative. The decision support automatically supply the corresponding values. This information includes for each alternative some rough estimates of the effort required and the complexity associated with that alternative. It also includes the availability of the resources to carry out the implementation of each alternative. The estimated values of these attributes are represented by the following:

1. ***Effort***: represented by the number of the new PSDL modules required to resolve the issue by taking that alternatives$_i$_*New*). We explained in the previous chapter how this value is computed.

2. ***Complexity***: represented by the number of the affected PSDL modules (*P*$_i$_*Affected*). We explained in the previous chapter how this value is computed.

3. ***Resources Availability***: represented by the availability of designers of the required expertise field and level to carry out the design and implementation of the selected alternative. This is computed by searching the designer's pool in the design data-base for objects that meets the requirements.

A session in this decision making process proceeds as follows:

1. The system builds a structure of the system using the following information:
   - The subject issue.
   - The set of judgement criteria (organized hierarchically if they constitute more than one level of the problem hierarchy).
   - The available alternatives.

2. The system links the issue, criteria, and alternatives with the proper links according to the Q-IBIS.

For each of the participating stakeholders, the system presents:

3. An instance of the problem hierarchical structure linked to the stakeholder object. The instance shows the index and the content of each level of the hierarchy. The minimum is one level for the criteria of judgement (general concerns) and another for the available alternatives plus the root.

4. Amplifying information for each alternative concerned with:
   - The requirement components affected, the anticipated effect, and any new requirement components to be added.
   - The computed values of the effort, complexity, and resources availability.
5. A user interface that allow the stakeholder to:
   - Conduct the pairwise comparison of the criteria that reflect their relative importance with respect to the debated issue.
   - Conduct the pairwise comparison of the alternatives that reflect their relative importance with respect to each of the criteria.
6. The scale to be used in the comparison process with enough interpretation.
7. A user interface that allow the stakeholder to informally express his argument, assumptions, or any other concerns

Enabled by this support, the stakeholder compares the criteria in pairs. Once he indicates the end of his comparisons, the system performs the intermediate computation to calculate a priority vector and a consistency index for the criteria that reflect that stakeholder judgement. The same actions are then repeated for the alternatives with respect to each of the criteria. The final result is a priority vector and a consistency index that prioritize the available alternatives with respect to the issue under consideration.

The results of the above process are used to automatically fill the attributes of the QI-position of the stake holder. The object of the stakeholder is then linked to this QI-position and QI-argument of that stakeholder using *has* link.

When all stakeholders are done, the system combines their judgement into one group position as wes explained (see page 113). The group is presented with an QG-argument object to record any informal information. The group then reviews the selection of the automated process, adjust it (if necessary), specify a deadline, and approve the outcome. This outcome is a change request to be implemented.

VI. CONCEPTUAL DESIGN OF THE DATABASE

This chapter discusses the conceptual design of the physical data storage that supports the requirements capture and evaluation. The choice of an individual implementation strategy should not affect the integrity of this conceptual design. Issues to be considered here are identifying the process entities, their properties, and the relationships that tie these entities. Entities are mainly derived from the conceptual model elements discussed in Chapter III.

A. DESIGN CONSTRAINTS

The design of the data storage of the requirements analysis and evolution system is not only constrained by functional requirements, but also by requirements dictated by CAPS. Categories of requirements that must be met by the system are as follows:

1. General Constraints

According to the data characteristics, the system must satisfy the following basic requirements which involves the background requirements:

1. For each prototype, the data storage should store the following pieces of information each with its relevant attributes and relationships. The storage is for both current as well as the historical values (if applicable) to support recording the rationale of the process:
    - Stake holders: this include individuals directly involved in the process either from the design team or the customer side.
    - The evolution graph with all its elements and relationships. This includes the following:
        - The set of software components: criticisms, issues, requirement components, PSDL components, and implementation components.
        - The set of analysis and design steps.

- The different types of edges that represent the relationships among the system parts as we discussed in Chapter III.

2. There is an extensive amount of specialization inheritance in the structure of the conceptual model which requires that the database schema design should support an object-oriented database system.
3. In modelling ill-structured real world problems, analyzing them , and creating design artifacts based on the modelling is better represented by a set of complex objects along with their relationships. This requirement is best served by having the database schema design support an object-oriented database system.
4. The mapping between reality, our conceptual model, and the prototyping model (the underlying software development model) is better served by designing the database schema to support object-oriented database [75].
5. Abstraction is a good way to hide irrelevant details. Therefore it is a thinking style that should be used to assist in comprehending customer needs more accurately. Object-Oriented modeling provides us with better abstraction facilities than other data modeling approaches. This requirement necessitates that the database schema design should support an object-oriented database system.
6. Because an object-oriented requirements specification is easier to understand by a customer than a functional specification, a database schema design should support object-oriented database.
7. The design database repository should allow managers, analysts, and designers (through a chain of responsibilities) to directly maintain and manipulate the repository content.

2. Constraints Imposed by CAPS

Since the data repository will be used to support an integral part of CAPS, this repository should meet the CAPS's specific requirements and constraints [67]:

1.The repository must be accessible in the UNIX based workstation, such as Sun or the compatible platforms.

2. The repository must be accessible in X-windows or the compatible windows environments.

3. Functional Constraints

3. The repository should support the basic database manipulation functionality, such as append, delete, modify, browse, backup, etc.
4. To track the evolution of software components given by our model graph over time, the database repository should provide facilities for multi-versions management.
5. The repository should support teamwork by providing mechanisms for concurrency and management of private and shared space.
6. The repository should be flexible in providing persistence for any kind of data structure because of the multi-type attributes the repository is expected to provide storage management to. This requirement also necessitates that a database schema should be designed to support object-oriented data modeling [60].

B. DATABASE SCHEMA DESIGN

The most important and difficult task for many database applications is the database design, often referred to as a *data model* or *schema* design [43]. It is clear from the above section that satisfying many design requirements, our schema should be designed to support an object-oriented data model. Such a model supported by the appropriate object-oriented database engine maps our conceptual model easily and naturally to the repository domain while keeping the implementation strategy independent on the schema design. In an object-oriented database design, the classical notion of a database schema is replaced by that of a set of classes or types [56] along with their attributes (properties), associations, and operations (methods). In the following subsections we discuss these issues and elaborate on some design decisions.

1. Types

All types of our schema are Abstract Data Types (ADTs). An ADT consists of a value set and a set of primitive operations that work on the value set. The instances of the value set are called the instances or objects of the type. An ADT can be either *mutable* if it has a an internal memory (state), or *immutable* if it does not. Instances of mutable types can be created, modified and destroyed by its primitive operations. Instances of immutable type can not be changed; a new instance is created whenever some change is needed.

In our schema we have both mutable and immutable types as shown in Table 6.6. It is quite reasonable to have the *Human* type as mutable and *Version* type and its subtype immutable. *Step* could be either mutable or immutable depending on the intended use. In our current design we chose to make *Step* type mutable. In an application where audit trails and/or contingency planning are required for process assessment or studying different design alternatives through e.g., design replay, the type *Step* should be immutable.

The schema types do not include a description for some of our Q-IBIS types. Specifically Alternative, Criterion, Position, and Argument are not included. The reason is that these types need a through examination regarding two points: representing them as attributes of existing types (e.g., issue) or as independent types. The second point is concerned with treating them as mutable or immutable types if they are represented as independent types.

Mutable Types	**Immutable Types**
Object	Version
Human	Criticism
Domain_Relation	Issue
Versioned_Object	Requirement_Component
Component_Reference	PSDL_Component
Step	Implementaion_Component

TABLE 6.6. Mutability of the Schema Types

2. Schema Type Hierarchy

As shown in the type structure of the schema shown in Figure 6.1, the schema contains fourteen types. Each of these types represents an element in the dependency graph of our conceptual model discussed in Chapter III. except *Object* type. *Object* type is the implementation vehicle of *persistence* in the object_oriented data model of ONTOS DB which we use as the object-oriented database engine. An object is persistent if it has life time that can extend beyond that of the process in which they are created.

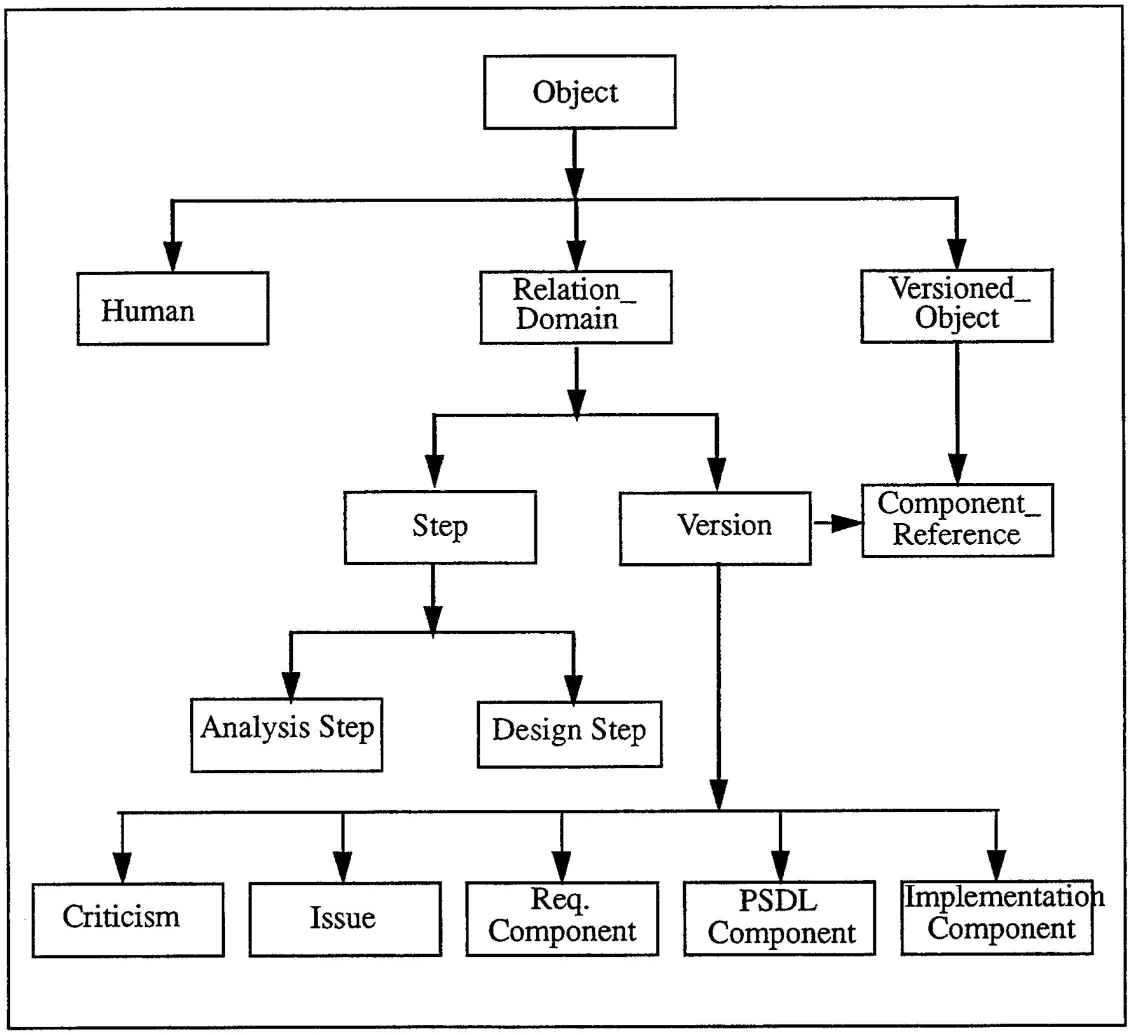


Figure 6.1 Schema Type Structure

All instances (objects) of our schema types are ***persistent***. The notion of persistence is central to the design and use of databases in general. The process of handling persistence in traditional database models (e.g., relational model) often involves conversion between the data structure used in the programming language and that used in the database. This adds the burden of writing code that translates between the disk resident representation of data and the in-memory representation used during execution. In object-oriented database the situation is different. No special constructs or very few such constructs are needed for storing and retrieving persistent data from the object-oriented database [75].

C. SCHEMA DESCRIPTION

In the following subsections we give the relevant attributes and operations defined over all types of the schema. We also discuss some design decisions and alternatives and highlight some notions. All types have an operation called *Create_Type* where Type is one of the eight schema types. This operation constructs a new instance (object) of the type and gives initial values to its attributes.

The underlying object-oriented database engine we selected is ONTOS DB database which is an object oriented database and uses C++ as a data definition and manipulation language (DDL, DML). Persistence allows a C++ object to be stored and retrieved from an ONTOS DB database.

1. Type Object

The type *Object* is the most general type. All other types are directly or indirectly subtypes of type object. All instances of this type and all its subtypes are persistent. This type is predefined in ONTOS DB to implement object persistence. Persistence of objects can be implemented in different ways [13]:

1. *By Typing*: we can declare that some types are persistent, thus every instance object of that type is persistent.
2. *By Reachability*: if an object is connected , through direct or indirect reference, to

some persistent root, then the object is persistent.

3. By Storing: if there is a persistent space, then every object explicitly bound into this space is persistent.

4. *By Object Indication*: there can be some parameters associated with the object which indicates whether the object is persistent or not

ONTOS employs the second approach for implementing object persistence where the *Object* type is the root of all persistent types. It is required that any persistent type be directly or indirectly a subtype of the *Object* type. Any type that is directly or indirectly derived from *Object* inherits all the capabilities that makes it persistent.

Table 6.7 and Table 6.8 summarize some relevant attributes and operations provided by the *Object* type. *putObject* saves an object into the database; *deactivates* it. *lookupObject*: looks up an object in the database and brings it into the application cache; *activates* it. *DeleteObject* from memory: deletes the in-memory copy and retains the DB copy. The names of these operation are given here abstractly; refer to [90] for exact names and signatures.

Attributes	
Name	Type
ObjectName	String

TABLE 6.7. Attributes of Object Type

Operations	
Operation	Effect
putObject (into DB)	Stores the object in the database with the option of deleting or retaining the in-memory copy
lookupObject	Retrieves the object from the database using its name and activates a copy in main memory
deleteObject (from DB)	Deletes the object from the database and deallocate the memory space
deleteObject (from memory)	Deletes the in-memory copy of the object with the option of deallocating memory space. The database copy is not affected
set ObjectName	Gives the object a name. The default is NULL
get ObjectName	Retrieves the name of an object

TABLE 6.8. Operations of Object Type

2. Type Relation_Domain

The Relation_Domain subclass is a general type used to realize the different binary relations embedded in the other types of the schema. It is needed only because ONTOS DB database does not directly support association between objects. Refer to Figure 6.2 that depicts an Entity-Relationship diagram showing an abstracted view of the relationships in the schema. Most of the superclass-subclass associations (inheritance) are shown in Figure 6.1 and are not repeated here. Also associations of the type *Version* are inherited by the different software components and was discussed and depicted in Chapter III.

Instances of the Relation_Domain type are directly inherited by the "Step" and "Version" types. The Relation_Domain subclass provides each of its subtypes with two instances for each embedded binary relation found in every subclass. One instance represents the forward direction of the relation, and the other represents the reverse direction (the inverse relation).

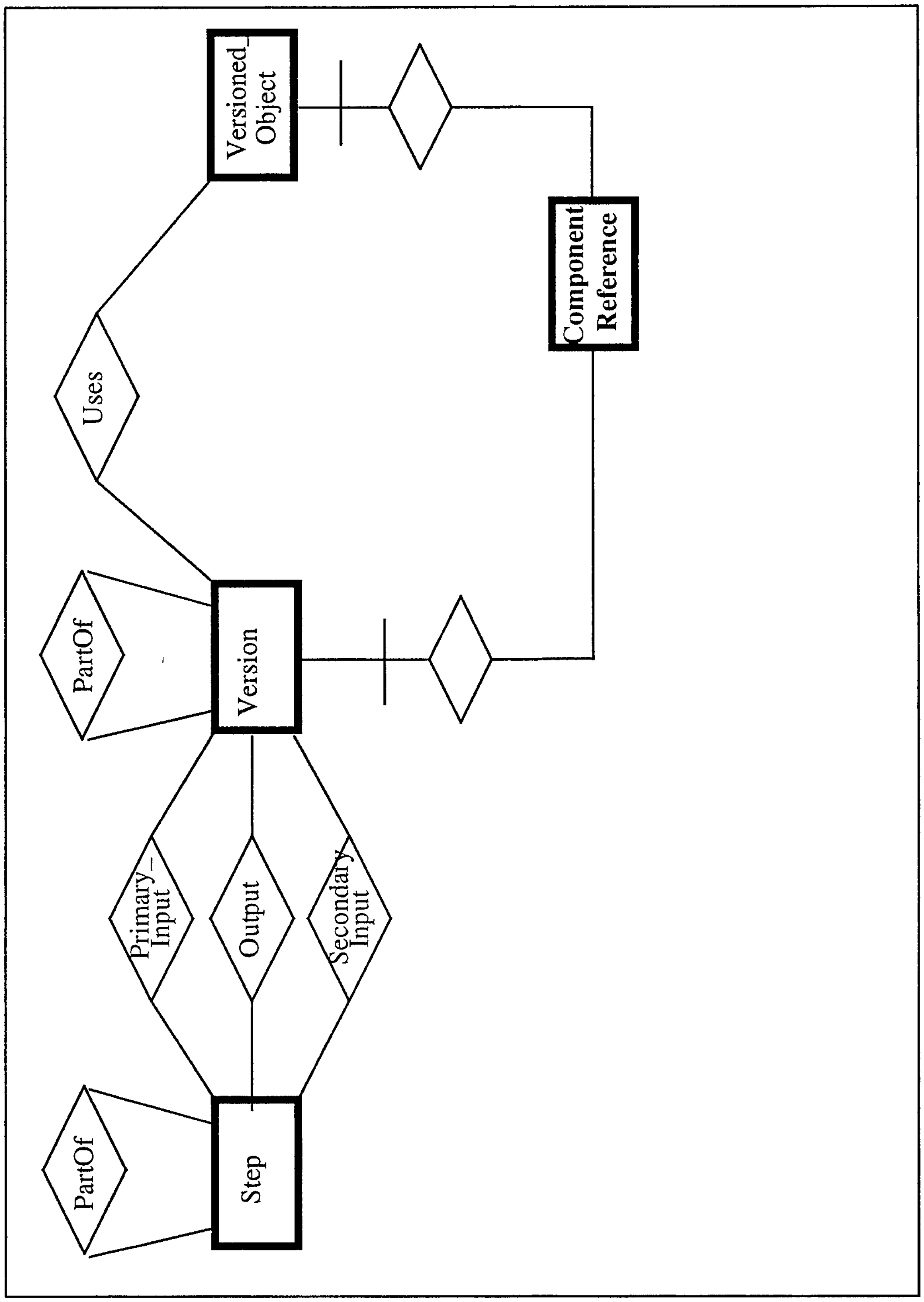


Figure 6.2 Abstracted E-R Diagram for the Schema Relationships

Each of the embedded binary relations relates an object from the relation domain type to a set of objects of the same or different type of the relation codomain. The inverse relation provide the same representation with the roles switched between the domain and the codomain. For each direction, the Relation_Domain provides a container for the set and a group of operations that work on it. This group provides for adding or removing a codomain object form the relation, computing the cardinality of the codomain objects in the relation instance, and finding out whether a codomain object is directly or transitively related to a given domain object.

The Relation_Domain subclass also provides two iterators for each direction of the relation. The first iterator is the *direct* iterator which iterates through the codomain objects of the relation and generate each. The second one is the *transitive* iterator which does not only iterate through and generate the directly related codomain objects, but also does the same recursively for all objects related by the same relation instance to each codomain object viewed as a domain object and has other objects related to as codomain objects. For example assume that an object x is related to an object y by the Relation_Domain instance R where x is from the domain of R and y is from the codomain of R (denoted as xRy). Also assume that y is related to a third object z; i.e., yRz. The transitive iterator provided by the Relation_Domain instance inherited by the object x generates y as well as z and any other object in the transitivity chain (the Kleen Star). Refer to Chapter VII. for implementation details and constraints of the transitive iterator. Both iterators; the direct and the transitive, are abstracted from a set of primitive operations that allow for creating, re-initializing, and destroying the iterator. They also allow for generating the next element as well as querying whether there are still more elements to be generated.

Macros are used to generate the Binary_Realation instances required to represent the embedded Relation_Domains in each of the types that inherits it. The reason for using macros are as follows:

1. Partially, because the version of ONTOS DB we are using does not support the new C++ template constructs which allow the development and reuse of generic types that can be instantiated to provide the same functionality for different types supplied as generic parameters.

2.Even with its availability, the C++ template constructs is not be flexible or enough to accommodate the generic scale we are willing to represent. We are not only seeking generic types as provided by C++ templates, but in addition we are seeking generic identifiers and naming to also verbally as well as functionally represent all binary relations embedded in different types of the DB schema.

The attributes and the behavioral operations provided by the Relation_Domain type are summarized in Table 6.9 and Table 6.10. The attributes and operations are given in a generic way and are instantiated appropriately by different types that need so. The instantiator provides a subject_role and object_role names for both directions of the relation. The subject_role name is used mainly in naming the instance and the object_role is used in the rest.

Attributes	
Name	Type
Object_role	Set{Objects}

TABLE 6.9. Attributes of the Relation Domain Type

Operations	
Operation	Effect
"add_"object_role	Adds an element to the codomain elements set
"remove_"object_role	Removes an element from the codomain elements set
"cardinality_of"object_role	Returns the number of the elements in the codomain set
"is_direct_"object_role	Querying whether a given object is currently related to a given domain object
"is_transitive_"object_role	Querying whether a given object is currently transitively related to a given domain object
Direct_Codomain_iterator	A set of primitive operations that implements a direct iterator.
Transitive_Codomain_iterator	A set of primitive operations that implements a transitive iterator

TABLE 6.10. The Operations of the Relation_Domain Type

As an example assume we are representing *Uses* relation between a software component and another set of components (either of the same or different types), the forward relation has *user* and *usee* as the subject and object roles respectively. The inverse relation has both names switched. The instance name is *user_domain* and *usee_domain* for forward and inverse directions of the relation respectively.

The attributes and operations that the instance of the forward direction of the *uses Relation_Domain* has are shown in Table 6.11 and Table 6.12 below with the proper indicative naming.

Attributes	
Name	Type
Usee	Set{Version}

TABLE 6.11. The Attributes of an Example instance of Relation_Domain Type

Operations	
Operation	Effect
add_usee	Add usee to the codomain set of the user object
remove_usee	remove usee from the codomain set of the user object
cardinality_of_usee	calculate and return the number of the usee objects currently in the codomain set
is_direct_usee	Querying whether a given usee is directly related to a given user object
is_transitive_usee	Querying whether a given usee is transitively related to a given user object
Direct_usee_iterator	A set of primitive operations that implements a direct iterator that works on the usee set directly related to a given user object
Transitive_usee_iterator	A set of primitive operations that implements a direct iterator that works on the usee sets transitively related to a given user object

TABLE 6.12. The Operations of an Example instance of Relation_Domain Type

The embedded binary relations found in other types and is represented by instantiating the Relation_Domain Type are as follows

1. Step:
   - *Part_Of*
   - *SubSteps*
   - *Primary_Input*
   - *Secondry_Input*
   - *Output*

2.Version
   - *PartOf*
   - *SubComponents*
   - *Uses*
   - *UsedBy*
   - *Affects*

- *AffectedBy*
- *PosedBy*
- *Input_For*
- *Secondry_Input_For*
- *Output_For*
- *Predecessor*

3.Human

- *Poses*

For each of these embedded binary relations there are two instances of the *Relation_Domain* type. The operations required for constructing, updating, and manipulating each relation in both directions is readily available. Other required operations may also be synthesized from the already available primitive operations.

For example, for *Substeps* relation, operations like *add_substep*, *remove_substep*, *cardinality_of_substep*, *is_direct_substep*, and *is_transitive_substep* is readily available to the forward direction of the relation with the appropriate naming. As an example for a computed operation is finding the top level step of a given step. This operation uses the iterator provided by the *PartOf* relation of the given step to iterate through the related objects (Steps) and return the object that has null parent.

Another important issue related to the discussion here is maintaining any one instance of the *Relation_Domain* type consistent in both direction. We accomplish this requirement by keeping an invariant that ensures whenever applying any *destructive* operation (e.g., add or delete) on any direction of the relation, the corresponding operation defined over the other direction is triggered automatically. For example adding a subcomponent *X* to a parent component *Y* automatically triggers the operation defined on the *PartOf* instance of *X* to add *Y* to the parents of *X*. In this way both the relation and its inverse are kept consistent all the time.

3. Type Human

Type *Human* represents persons involved in the process either from the customers or the design and analysis team. The relevant information about an individual involved in the process are name, roles, organization, skill, and availability status. The *name* attribute is directly inherited from the immediate superclass *Object*. *Roles* set for the design and analysis team is known and more specific than for customer individuals. For design and analysis team this set includes: designer, analyst, and manager. For customer individuals, it varies from one application to another. Therefore it is considered application defined.

Organization attribute refers either to individuals from the prototype team (designers, analysts, or managers), or individuals from the customer side. In the latter case this attribute carries the name of the organization financing or sponsoring the project. It is represented as a set of strings instead of a single string to account for cases where the project is financed or sponsored by more than one organization.

For the prototype team, a *skill* represents the different fields of expertise for a team member and the member skill in each of this fields. For a customer individual it represents that individual's skill for each role he plays in his organization. For example, *John Smith* has the roles "*Sales*" in *ALM company* and his skill in *"market analysis"* is *Medium*, and in "*Company priority*" is *High*. The skill attribute is represented by a map from strings to enumeration. We chose the domain of the map to be a string instead of another enumeration to avoid restricting the set of expertise fields.

The attribute *Status* is an enumeration telling whether an individual is currently available or busy. This attribute makes more sense for individuals that belong to the prototype team and is very important for planning and task assignment activities. For customer individuals it can assist in planning the prototype demonstrations by knowing the availability status of the customer individuals. However in the latter case, this attribute is required to model more information to elaborate on the availability periods. This elaboration is useful also in the case of the team individuals for planning purposes and will be more effective if a reason is attached to the non-availability status of an individual (e.g., specifying whether a designer is busy assigned to another task or on leave, etc.).

In addition to the set of operation defined over the above attributes, a set of other primitive operations were added to abstract an iterator to iterate through the individual elements and generate each person. The latter set of operations allow for constructing and initializing the iterator as well as freeing the memory it is allocated. The iterator provides the check whether there are still more elements to generate and generate each. Type *Human* directly inherits from Object and has no subclasses. *Human* attributes and operations are shown in Table 6.13 and Table 6.14.

Attributes	
Name	Type
Roles	Set{string}
Organization	String
Skills	map[(exp_field: string) --> enumeration]
Status	Enumeration
Poses (inherited)	Poses_Domain (instance of Relation_Domain)

TABLE 6.13. The Attributes of the Human Type

Operations	
Operation	Effect
set_organization	Assigns a new value to an individual's organization
get_organizatio	Retrieves the value of an individual's organization
add_role	Adds a role to an individual's roles set
remove_role	Removes a role from an individual's roles set
cardinalty_roles	Returns the number of roles of a specific individual
set_status	Assigns a new value to an individual's status
get_status	Retrieves the value of an individual's status
find_designer	looks up and retrieves an individual's from the data-base using his name
add_skill	Adds an expertise field and an associated expertise level to an individual's skills map
remove_skill	Removes one of the expertise fields and the corre-sponding expertise level from an individual's skills map
change_skill_level	Changes the skill level associated with a given expertise field for an individual
generate_designer (iterator)	A set of primitive operations that implements an iterator that iterate and generate all individuals in the database

TABLE 6.14. The Operations of the Human Type

4. Type Versioned_Object

Type Versioned_Object encompasses a sequence of variation lines of one object identified collectively by an object id. A variation line abstracts a sequence of versions of the same versioned object. Each of the sequence elements is a frozen state of the object which either evolved from the previous version, an initial creation, or as the result of a decomposition of a composite versioned object. An evolution step is required for a version to evolve from an existing version or to be created as a decomposition of a parent component. For a version to evolve from an existing version, the existing version must be one of the inputs of the evolution step and the new version must be one of the outputs for the step.

A variation works as a building block for the versioned object type. Initially a versioned object has only one variation line. Each time a new version is created is added to the end of that variation line until the need arises for splitting a new variation line. A new variation line is split only if every predecessor (to be defined in the context of the "Version" type) of the newly created version is either:

- Not current, (see Figure 6.3), or
- Has a gap in version number (see Figure 6.4).

The first case is illustrated by the following evolution graph

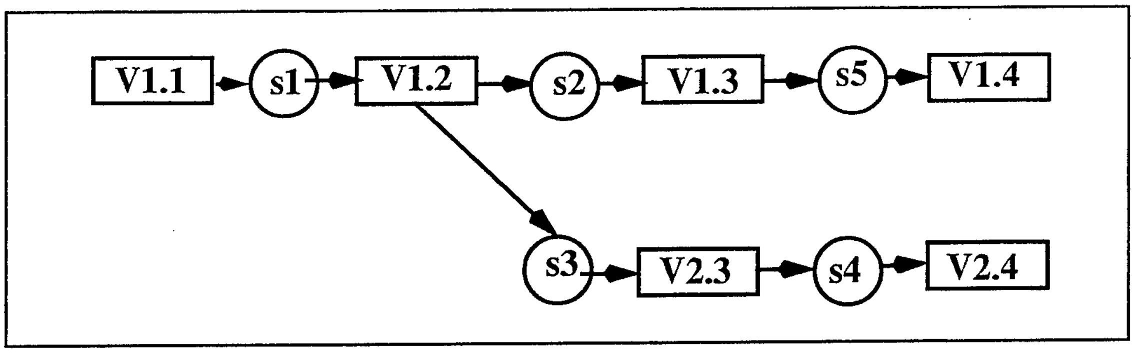


Figure 6.3 New Variation Split: Case1 [69]

The current version of a variation is the version with the maximum version number among all versions that belong to the variation line. In the figure above, the current version of the variation V1 is V1.4 but step s3 has V1.2 as a primary input. This situation requires the split of a new variation V2. The new version V2.3 will have V1.2 as a predecessor.

The second case applies to merge cases as shown by the following evolution graph where the changes in V1.4 and V2.4 are merged by the step s6 resulting in the new version V1.5 along the first variation line. After that, step s7 is merging V1.4 and V2.3 and none of them is current, so a new variation V3 is split.

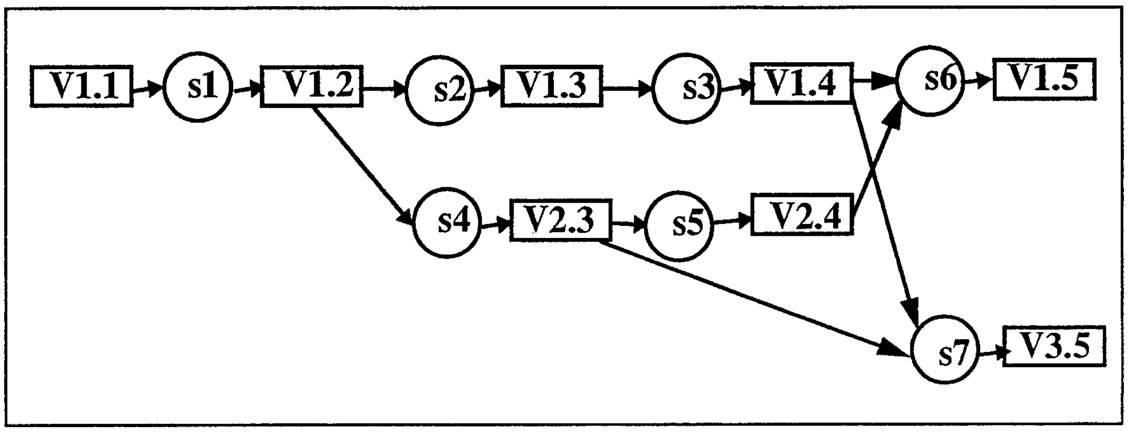


Figure 6.4 New Variation Split: Case2 [69]

Another question related to the merge case is what existing variation the merge result belongs to if it does not split new variation. The new version of the merge result will belong to one of the existing variation lines that must satisfies the following three conditions simultaneously:

- The version number of the merge result must be maximum value.
- The version number of the predecessor version on that variation line + 1 = the version number of the merge result.
- The predecessor version (node) must be the current version of that variation line at the merge time.

One set of the attributes and operations defined on the "Versioned_Object" type works directly on a given "Variation". Operations that will be used the most for a Versioned_Object is adding a version to the end of a variation line or finding a version given its version and variation numbers.

The *Description* attribute is a string summarizing the functionality or any other specific semantic related to the versioned object. It is specially useful for newly proposed requirements and PSDL components as part of resolving an issue. If the alternatives encompassing these components are finally rejected as a result of the resolution, these components are not added to the configuration. At this point the rejected components are just drafts; only a few attributes are assigned meaningful values, *Description* is one of them. This attribute makes future references to the rejected components more fruitful. We give another related attribute in subsection 6.

The Type “Versioned_Object” also includes a sequencer for controlling the evolution process of the object. It directly inherits from “Object” type and has the type “Component_Reference” as a subclasses. The attributes and operations of this type are shown in Table 6.15 and Table 6.16 respectively.

Attributes	
Name	Type
version_map	map[tuple[ver,var: id#] --> version]
length	map[(var: id#) --> natural]
current_var	var_id
Object_Sequencer	Sequencer
Description	String

TABLE 6.15. The Attributes of the Versioned_Object type

Operations	
Operation	Effect
find_version	Finds and returns a version given its version and variation numbers
find_current_version	Finds and returns the current version in a variation given the variation number of the variation it belongs to
add_version	Adds a version to the end of a variation line. It becomes the current version of that variation line

TABLE 6.16. The Operations of the Versioned_Object type

Operations	
Operation	Effect
set_current_var	Sets the current variation of a versioned object.
get_current_var	gets the current variation of a versioned object
find_recent_common_version	Finds and returns the version with the maximum version number common to a merge result

TABLE 6.16. The Operations of the Versioned_Object type

5. Type Version

As was indicated earlier the "Version" type represents a frozen state of an object. It abstracts the idea of an evolving software component. In our context it is used to represent Criticisms, Issues, Requirement, and PSDL Components. A PSDL component represents in principal a PSDL operator (composite or atomic) which has one specification and another implementation component linked together by 'uses" relation. The type version is general enough that can represent (either as is or through specialization) any other kind of software components e.g., test cases.

A version belongs to one or more totally ordered variation line. Its variation_id and version_id determines which Variation it belongs to and where it is located in the variation. Each version has a predecessor which is a set of versions that contains all versions in all variation lines that proceeds the version. The predecessor set contains at least one version except for the first version in the first variation termed *base_version* that has an empty predecessor set. For example in Figure 6.4 the predecessor set of version V3.5 includes V1.4 and V2.3 and the predecessor set for V1.1 (which is the base_version too) is empty.

Type Version inherits from instances of Relation_Domain type which is very useful in representing the embedded binary relations to be kept by the type Version. It is the superclass for Criticism, Issue, Requirement_Component, and PSDL_Component. The "type" attribute relates a version to the type of software components it represents.

Dependencies among these different software components can be traced down and up employing the "Used_By" and "Uses" Relations. The "used_by" instance of the Relation_Domain type inherited from "version" type links a requirement component to a PSDL component and can be used in the trace from requirement down to the implementing PSDL component(s). The other direction of the trace is given by the "uses" instance of "Relation_Domain" type defined on a PSDL component. The "required_by" identifier string of the PSDL component can be filled from the requirement component that PSDL component uses.

In Table 6.17 and Table 6.18 below we give the attributes and operations defined over "Version" type. We do not give operations specific to the embedded binary relations since they have been elaborated in detail earlier. Another important attribute is the "Content" which provide a container along with some operations for a set of text files along with their names associated with the version.

Attributes	
Name	Type
version_id	Natural
variation_id	Natural
time_created	time
created_by	String
PartOf (inherited)	Part_Of_domain (instance of Relation_Domain)
Subcomponents (inherited)	Subcomponent_domain (instance of Relation_Domain)
uses (inherited)	usee_domain (instance of Relation_Domain)
usedBy (inherited)	user_domain (instance of Relation_Domain)

TABLE 6.17. The Attributes of the Version type

Attributes	
Name	Type
Affects	affecting_domain (instance of Relation_Domain)
AffectedBy	affectedBy_domain (instance of Relation_Domain)
PosedBy	poser_domaain (instance of Relation_Domain)
Primary_Input_for (inherited)	Primary_Input_for_domain (instance of Relation_Domain)
Secondry_Input_for (inherited)	Secondry_Input_for_domain (instance of Relation_Domain)
type	Enumeration
Output_Of (inherited)	Output_Of_domain (instance of Relation_Domain)
content	map[(file_name: string) --> string]
predecessor	Predecessor_domain (instance of Relation_Domain)

TABLE 6.17. The Attributes of the Version type

Operations	
Operation	Effect
set_version_id	Sets the version id of a given version
get_version_id	Gets the version id of a given version
set_variation_id	Sets the variation id of a given version
get_variation_id	Gets the variation id of a given version
set_time_created	Sets the time the version was created
get_time_created	Gets the time the version was created
set_created_by	Set the name of the analyst or the designer who created the version
get_created_by	Get The name of the analyst or the designer who created the version

TABLE 6.18. The Operations of the Version type

Operations	
Operation	Effect
set_type	Sets the type of the version **(should not be defined as an attribute of versioned_object???)**
get_type	Gets the type of the version
add_text_file	Adds a text file to the content map of the version.
delete_text_file	Deletes a text file to the content map of the version
get_file_names	Gets the file names of the content of the version
find_text_file	Finds and returns a text file that belongs to the content map of the version. The file is given by its name.
cardinality_text_files	Returns the number of text files the version has.

TABLE 6.18. The Operations of the Version type

6. Type Component_Reference

When an analysis or evolution step is proposed and before it is assigned to a designer, it is known what a versioned_object it applies to. However, it is not known what specific versions of which it applies to. Therefore from the time proposed until its primary inputs are bound to specific versions, a step primary input is tied generically to a versioned_object given by its id. Once the specific primary input set becomes known, the step binding is changed to those specific input versions. For this reason the Component_reference is used to play both roles; the generic and the specific one. It inherits from versioned_object type to play the generic role and from version type to play the specific role.

This type also plays an important role for the newly created versioned objects. As we explained in Chapter III., new requirements and PSDL components may be proposed in the process of exploring and analyzing the impacts of available alternatives for resolving an open issue. The binding of these newly created components is not known until after the issue is resolved by choosing one or more alternatives. The components affected by the taken alternative is then bound to specific versions. The other components are not bound but remain in the database as part of the analysis and design history. Since these latter components do not and may not represent a part of the current or the future configuration, representing them by component reference objects is a reasonable solution. To differentiate between component reference objects in this case and other cases, we associate one attribute with this type as a discriminator. The *IsUnderAnalysis* attribute is of type *Boolean* that has the value *True* if an object of the component reference type is not part of the configuration, and *False* otherwise.

7. Type Step

Type Step represents a design or analysis step. It directly inherits from six instances of the Relation_Domain type. Those instances are used to represent six embedded binary relations inside the "Step" type as was discussed previously. In addition to those inherited, type step defines more properties and operations of its own. Type "Step" has two subclasses derived from: analysis and design steps. These two subclasses are different in their semantic content and their approval rules as was discussed in Chapter III. Other than that both subclasses have basically the same set of attributes and operations. Although both types can be easily differntiated by the type of the input set to the step, the attribute type is added to the parent type "Step" to easily differentiate between both. Having this attribute can also help clustering each subtype objects in storage which may contribute in speeding up the process. Table 6.19 and Table 6.20 contain the relevant attributes and operations defined on the type "Step". Again we do not include the operations defined over the embedded relations.

Attributes	
Name	Type
step_id	Natural
step_type	Natural
priority	Natural
designer	String
required_skill	Enumeration
status	Enumeration
start_time	Time
finish_time	Time
duration	Time
estimated_time	Time
deadline	Time
date_created	Time
date_of_current_status	Time
Part_Of(inherited)	parent_domain (instance of Relation_Domain)
Substeps(inherited)	substep_domain (instance of Relation_Domain)
Primary_Input (inherited)	primary_input_domain (instance of Relation_Domain)
Secondry_input (inherited	secondry_input_domain (instance of Relation_Domain)
Output	Output_domain (instance of Relation_Domain)

TABLE 6.19. The Attributes of the Step type

Operations	
Operation	Effect
set_step_id	Sets the step id
get_step_id	Gets the step id
set_step_type	Sets the step type
get_step_type	Gets the step type
set_priority	Sets the step priority
get_priority	Gets the step priority
set_designer	Gets the designer to be assigned the task associated with the step activity
get_designer	Sets the designer to be assigned the task associated with the step activity
set_required_skill	Sets skill required to perform the task associated with step activity
get_required_skill	Gets skill required to perform the task associated with the step
set_status	Sets the step status
get_status	Gets the step status
set_start_time	Sets the step start time
get_start_time	Gets the step start time
set_finish_time	Sets the step finish time
get_finish_time	Gets the step finish time
set_duration	Sets the step duration
get_duration	Gets the step duration
set_estimated_time	Sets the step estimated time for completion
get_estimated_time	Gets the step estimated time for completion
set_deadline	Sets the deadline by which the step should be completed
get_deadline	Gets the deadline by which the step should be completed
set_date_created	Sets the date the step was crated (proposed)

TABLE 6.20. The Operations of the Step type

Operations	
Operation	Effect
get_date_created	Gets the date the step was crated (proposed)
set_date_of_current_status	Sets the date of the step current status
get_date_of_current_status	Gets the date of the step current status

TABLE 6.20. The Operations of the Step type

8. Other Types

The rest of the types in the schema represent different categories of software components encountered in the analysis and design process. These include criticisms posed by human individuals in response to the demonstration of the prototype, issues to be resolved to take into account the justifiable criticisms, requirement components, and PSDL components. All of these types are immutable even if only one version is archived for some of these types like criticisms. The need to make criticisms immutable is motivated by the idea of documenting this kind of the formal communication with customers intact as justifications for any subsequent changes in the requirements.

Attributes and operations required by any of these types are directly inherited from *Version* type; their immediate supertype. This is because the main difference among these types is the semantic difference in their textual contents.

D. CONCURRENCY CONTROL

One of the main requirements to be satisfied by our system, as was mentioned in section A.3 of this chapter, is to support teamwork which implies that the DBMS should support concurrency. Concurrency Control facility is provided by most DBMS to support data sharing so that multiple concurrent access to the data repository does not lead to an inconsistent state of the database. Providing concurrency control means that transactions are serializable: the result must be the same as some serial execution of the same transactions [60].

In our context, we provide some high- level serialization rules specially for analysis and design steps as was discussed in Chapter III. Additionally the ECS which our work extends its scope and capabilities, does provide automatic serialization based on management policy for design steps that does not permit starting a step before a preceding one completes. We also rely on the underlying concurrency control facility of the database engine to provide multiuser concurrent access to shared objects.

The object-oriented paradigm is very natural for modeling concurrent systems [75]. The most common approach object DBMSs use for concurrency control is *locking*. In its simplest form, locking an object prevents other transactions from using that object until the lock is released. The DBMS should also provide the capability to mange locks. Managing locks includes [60]:

- Granting locks to transactions on particular objects in response to their lock requests.
- Keeping track of which transactions hold which lock.
- Detecting when lock requests interfere with each other.
- Clearing locks when transactions release them.

Two types are normally used for concurrency in OODBMS [10]:

1. ***Read locks:*** Used by a transaction to prevent other transactions from committing a newer version to the database than the one it is reading until the lock is released. Several transactions can share a read lock.
2. ***Write locks:*** Used by a transaction to prevent other transactions from accessing an object because it is updating the value of the object. Write locks are exclusive nad are not shared with other transactions.

In the following subscetions we discuss concurrency control scheme and policies provided by ONTOS DB; the object-oriented database engine we choose as the underlying DBMS. The discussion is based on ONTOS DB documents [89], [90].

1. Concurrency Control Scheme in ONTOS DB

ONTOS DBMS uses locking technique to implement concurrency control. The underlying scheme is centered around the idea of a transaction. A transaction is an atomic unit of change; either all of the changes it contains are saved to the database or non of them is. This is necessary to keep the database state consistent. For this reason the changes to an object can only be saved to the database within the scope of the same transaction.

A transaction has access to the client cache of ONTOS DB objects on the application side and has access to the server cache. Only through the latter an application can access the objects in the DB either to retrieve (get) or to store (put). The client cache is the area of application memory where DB objects are stored. Objects brought from the server cache into the client cache are called "activated" or "in memory".

ONTOS DB employs three concurrency control policies (to be explained shortly). The choice among them is specified by three parameters given to OC_transactionStart() free function which starts a transaction. An application also has the freedom not to specify any relying on the default values. Those parameters determines the following:

- A conflict detection protocol for identifying the conflicts arising from concurrent attempts to access an object.
- A conflict response protocol for responding to conflicts arising from attempts to lock an object for reading or writing.
- A buffering protocol for buffering objects on the client side before they are output to the sever cache

a. Conflict Detection Protocols

ONTOS DB supports two conflict detection protocols

(1) **Readers and Writers do not Conflict (NoRWConflict):** This Protocol maximizes overall concurrency across all application accessing the database. It allows

processes to obtain ReadLocks (**not WriteLock**) on objects that have already been WriteLocked. The reader sees an earlier or later version of the object than the writer depending on which is serialized first. If a deadlock occurs, a preemtive abort occurs.

(2) **Readers and Writers Conflict (RWConflict):** Under this protocol the only concurrent access allowed is to readers on object that already ReadLocked. This is the **default protocol** provided by ONTOS DB. Note that writers conflict with writers in either protocol.

By default objects activated under NoRWConflict have RaedLock and objects activated under RWConflict protocol have WriteLock. This can be overwriden by specifying a lock type when activating an object by supplying the lock type as an argument to ONTOS functions responsible for activating objects (e.g., OC_Lookup()).

b. Conflict Response Protocols

The Conflict Response Protocol specifies a function to be called when a conflict occurs. Two protocols are available:

(1) **Wait on Conflicts:** waits until the lock is released by the locking process. This is the **default protocol** provided by ONTOS

(2) **Raise the WaitException:** allowing the application to take control after a conflict to retry the database or do some other work.

c. Buffering Protocols

Buffering protocol determines how frequent objects are transferred from the client cache buffer to the server buffer during the course of a transaction. Three options are available:

(1) **No Buffering:** each *put* call results in an immediate transfer to the server.

(2) **Default Buffering:** objects are buffered and sent in small groups to the server. Typically, a transmission is made after ten *put* operations.

(3) **Buffer Until Commit:** all objects are transmitted all at once when the transaction is committed.

2. Concurrency Control Policies

ONTOS DB supports three concurrency control policies. It should be noted that **under any of the three policies all the objects viewed by each transaction are from a consistent state of the database**. The three policies differ only on the degree of concurrency each provides and the consequences based on that provided degree of generosity in terms of transaction aborts likelihood:

a. Conservative Policy:

This is the **default policy** provided by ONTOS. It is implemented by *the Readers and Writers Conflict Protocol: RWConflict.* Under this policy, a process attempting to get a ReadLock or a WriteLock on an object WriteLocked by someone else or attempting to get a WriteLock on an object that already has a ReadLock, can not access the object and faces one of three fates:

- Wait until the object is unlocked.
- Abandon its attempt to access the object.
- Abort the transaction.

b. Time-based Policy:

This policy is implemented by using *No Read Write Conflict Protocol: NoRWConflict* and the default buffering (or no buffering). Under this policy the default locks on objects is ReadLock.

c. Optimistic Policy:

This policy provides the widest overall access to the database and accepts high risk of abort due to unresolved conflicts. It is implemented, like Time-based policy, by using the NoRWConflict detection protocol but uses Buffer Until Commit protocol for buffering.

3. Recommendation

For the design database, we recommend the use of the ***conservative*** concurrency control policy combined with the ***"wait on conflict"*** conflict response protocol. For the buffering we use the ***default buffering.*** Default buffering is a compromise between the other two buffering policies. The problem with the ***"No Buffering"*** is that it reduces the efficiency of transmission from the client (application) to the server. The server and the client are assumed to lie on different sites of a network as is often the case. The third policy, "***Buffer Until Commit***"***,*** has a different problem: it lacks timely information. This comes from the fact that the application can not get any information on lock conflicts until a ***put*** is actually made to the server. Other reasons for our choice are as follows:

1. Within the context of this policy, transactions are the least likely to be aborted due to unresolved conflicts.
2. The use of ***"wait on conflict"*** protocol reduces aborts to a bare minimum.
3. No deadlock is possible due to this policy. Deadlock is expensive to break in the other two policies. It is broken by preemption that leads to abortion.
4. Writing database schema and application code is easier in the case of this policy. This is because this policy and the conflict response protocol along with the buffering protocol we choose, are the defaults of the concurrency parameters of ONTOS DB. Therefore, the programer does not need to supply any of these parameters to the ONTOS free functions that require them. These functions use the default values in this case.

5. The other two policies require exact specification of the concurrency parameters which creates the potentials for unintentional mistakes resulting from the failure of supplying the proper combination of these parameters. This may lead to following a concurrency scheme other the one intended.

6. Although the level of concurrency provided by this policy is the least, we rely on high level serialization for analysis and design activities that makes this problem less sever compared to abortions of transactions.

7. Since this policy allows concurrent access to readers on objects that already Read-Locked on any level of concurrency, it still maintains the same concurrency level for read-only access. This is very suitable for search and view-intensive applications which conforms with the requirements of the software base in CAPS. Therefore the use of this policy assists in establishing standardization of the database use over all contexts supported by the database in CAPS.

VII. IMPLEMENTATION MODEL

A. THE DATABASE ENGINE

We have chosen ONTOS DB as a database engine for our design database. ONTOS is an Object-Oriented multi-user distributed database that includes standard database functionality as well as a specific support for objects. C++ is the language used to interface to ONTOS DB and is also used as a Data Definition and Manipulation Language (DFL, DML).

1. Why ONTOS DB

We have chosen ONTOS because it provides us with [89]:

1. A reliable persistent facility for C++ objects
2. Standard database capabilities and special support for objects.
3. A set of database and object-oriented classes to enhance the power of C++.
4. A set of useful tools for application construction.

Developing application using ONTOS requires writing C++ code to do the following:

1. Defining the classes that comprise the types of the database schema.
2. Implementing the classes defined in 1.
3. Implementing the overall application using the above classes and possibly the classes supplied in ONTOS DB.
4. Implementing a suitable user interface.

2. Data Manipulation Language Problem

Despite the fact that C++ is the DDL and DML language supported by ONTOS DB, many factors discouraged us from using C++ directly or at least at all levels of our implementation and using Ada instead:

1. Ada is the major implementation language for CAPS where our work is undergoing

within its context. Within CAPS, Ada is used for developing prototypes, reusable components, and the target application.

2. Ada is relatively easier to deal with in terms of the availability of good and stable compilers and run-time systems.
3. In CAPS a belief was built from actual experiences that ***debugging*** of C++ programs proved to be difficult, specially run-time errors. This increases the required development effort where speed is required for (rapidly!) designing and developing prototypes.
4. We have built a good deal of experience in CPAS with User interfaces using TAE [91] and coded in Ada. This resulted in a large volume of good and reusable fragments of Ada code that can be used partially to implement our user interface.

However ONTOS DB has no Ada binding, and the only language one can use with ONTOS (either as a DDL, DML, or application development language) is C++. Our solution to the problem is outlined below and is detailed in the rest of this chapter.

3. Outline of the Solution

The main theme of our solution to the above problem is to keep the C++ code as minimum as possible for defining and implementing the classes that model the schema types and to communicate with ONTOS in the low level. Ada does the rest in all other levels. This effectively means we build an Ada binding ourselves that allows us to use ONTOS as if it supports Ada. We experimented our approach using a design database mock up which we designed and developed as a template to be followed not only as part of our work but on any other similar applications even under different contexts. Refer to Appendix A for a detailed picture of the basic code comprising this mock up.

This template was made carefully and general enough to provide a complete methodology that can be used to provide an approximate Ada binding for ONTOS. Our methodology is explained in detail below.

B. AN ADA BINDING FOR ONTOS DB

One vital decision regarding our methodology to build an Ada binding for ONTOS is the binding boundary. It is clear that the schema definition and implementation must be developed using C++. This include the definition and implementation of some C++ classes that represent the schema types along with their attributes, relationships, and operations. The rest of the required application code uses the schema code along with ONTOS supplied classes and free functions to do the operations required by the application normally through a suitably developed user interface. Operations include general database transactions such as create, store, retrieve, and update objects. Therefore we divided the required code into three communicating layers where each layer provide services to the layer above. The lowest layer includes the C++ code defining and implementing the database schema as well as ONTOS supplied libraries of classes and free functions. The second layer above includes the code that implements the application logic including the database manipulation. The third layer above includes the code implementing the user interface. In our approach we decided to implement the second and third layers in Ada. Since both layers are implemented in Ada, there was no problem to let both communicate. The real problem was how the second layer (implemented in Ada) communicate with the first layer (implemented in C++). The approach we developed is explained below:

1. Motivation

Within CAPS many attempts were made to let Ada and C++ communicate smoothly. All previous attempts [37], [72] tried to use system calls to inter-communicate Ada and C++. System calls have the advantages of easiness to understand, implement, and requires minimal coding. However system calls are probably suitable in situations where we need to trigger an independent process from the currently running one. Independence here means that the process runs to completion without the need for inter-communication with the calling process.

Another related problem is that using system calls requires the use of files for parameters passing which is complex and inefficient. On the other hand, direct parameter passing between the two languages can result in very persisting kind of run-time errors. The difficulty comes from the fact that most data structures have different binary implementation in both languages. In the mean time for data to be passed among C++ and Ada programs, these data must have exactly the same binary implementation. There are some solution for the problem, for example by using Ada representation specification. Even the available solutions are complex, very error-prone, and sometimes do not work.

Given the above facts in addition to the fact that design database systems are highly active and in which process inter-communication is heavy, our goal was to device a new technique to achieve the following two objectives simultaneously:

1. The new technique will not use system calls method for realizing inter-communication between the two languages.
2. The new technique should solve the parameter passing problem.

2. A New Inter-Communication Methodology Between Ada and C++

The methodology we developed achieves the above two goals. It allows programs in both sides to communicate directly without the need for system calls. It also solves the parameter passing problem and thus bridges the representation mismatch between the two languages. The price paid is some extra, may be duplicate, code and potentially increase in code size.

a. Inter-Communication Controller (ICC)

The key idea of the new technique is to dedicate some code for controlling the traffic between both languages; let us call it *Inter-Communication Controller (ICC)*. The sole purpose of the ICC is to facilitate the communication between Ada an C++. Part of the code implementation of the ICC is a C++ code (to be abbreviated by ***CICC*** hereafter), and

The next task the analyst does is to acquire and synthesize a set of issues from the criticisms of the SHs. He does this using also the process history. The history for this task is approximated by the set of issues from the last demonstration. If one issue can be the generalization or the specialization of a previous one, this makes the resolution process much more easier by adapting the artifacts used to resolve it. The new issue may also has its solution in one of the abandoned alternatives for the old one. The new issue may contradict the resolution of an old one. These are some of the reasons for recording the history of the process; design rationale is not lost.

The issues synthesized from the refined set of criticisms are:

1. **Issue1**: *The undistinguished coexistence of current and obsolete tracks in the TDB is not recommended.*
2. **Issue2**: *The TDB must not allow the storage of duplicate tracks received from different sources (including local sensors).*
3. **Issue3**: T*he TDB must allow the user to filter the retrieval and display according to his focus.*

In this simplified case, it happened that the mapping is one-one between the criticisms and the issues. This is not the case in general though. One criticism may map to two or more issues and vice versa, i.e the mapping is many-to-many in the general case.

The analyst reviews the currently synthesized issues, compares them against the previous ones, and creates an object for each using the system. These objects are then linked to the step as its output. The analyst also establishes the link manually between the issues and the criticisms from which they were synthesized. At this point the portion of the new state of the dependency graph acquired so far is given in Figure 8.8. The design database stores the same image for this portion too.

the other is an approximate mirrored image of the first part but implemented in Ada (to be abbreviated by ***AICC*** hereafter). The structure of the whole application then is as follows:

1. Ada code in one side implementing the major part of the application and the user interface.
2. C++ code in the other side defining and implementing the schema.
3. The ICC in between.

The following parts can talk directly:

- The AICC and CICC.
- The CICC with the C++ code implementing the schema.
- The AICC with the Ada application programs.

b. Communication Protocol

For the purpose of illustration, assume that from the Ada side we are willing to change some attribute value call it *status* to become *busy* of some person object whose *name* attribute is supposed to have the string value "*John*". The communication between the system parts adheres to the following protocol at all levels:

1. The user interface is used to invoke the system and enter the required values ("*John*" and *busy* in our case).
2. The Ada application programs capture these values and passes it as parameters to the AICC.
3. The AICC abstracts the context object into a dummy pointer or pointers and passes it to the CICC along with any identifying information to be used for looking up the object in the database, in our case it is a pointer to the string "John".
4. The CICC communicates directly with the C++ code passing to it the identifying piece of information.
5. If the lookup operation is successful, the CICC, points to the object with the dummy pointer it received from the AICC.

6. The CICC sends back the pointer to the AICC.
7. The AICC sends the pointer again to the CICC along with the *status* new value, *busy.*
8. The CICC dereference the pointer to invoke, within the context of the pointed to object, the C++ class method *ChangeStatus(new_status)* to perform the required work.
9. If that required work is to return some value(s) back to Ada, a pointer to the resulting value is returned.
10. If any subsequent operations are to be performed on the same object, only steps 7 through 9 are repeated.

The point behind this excessive back and forth traffic is to avoid derefrencing C++ pointers in the Ada side, especially references to large and complicated data structures such as C++ objects. If it happens, the derefrencing in this case is the source of many dangerous errors. Figure 7.1 illustrates the idea of the ICC in between Ada and C++ code.

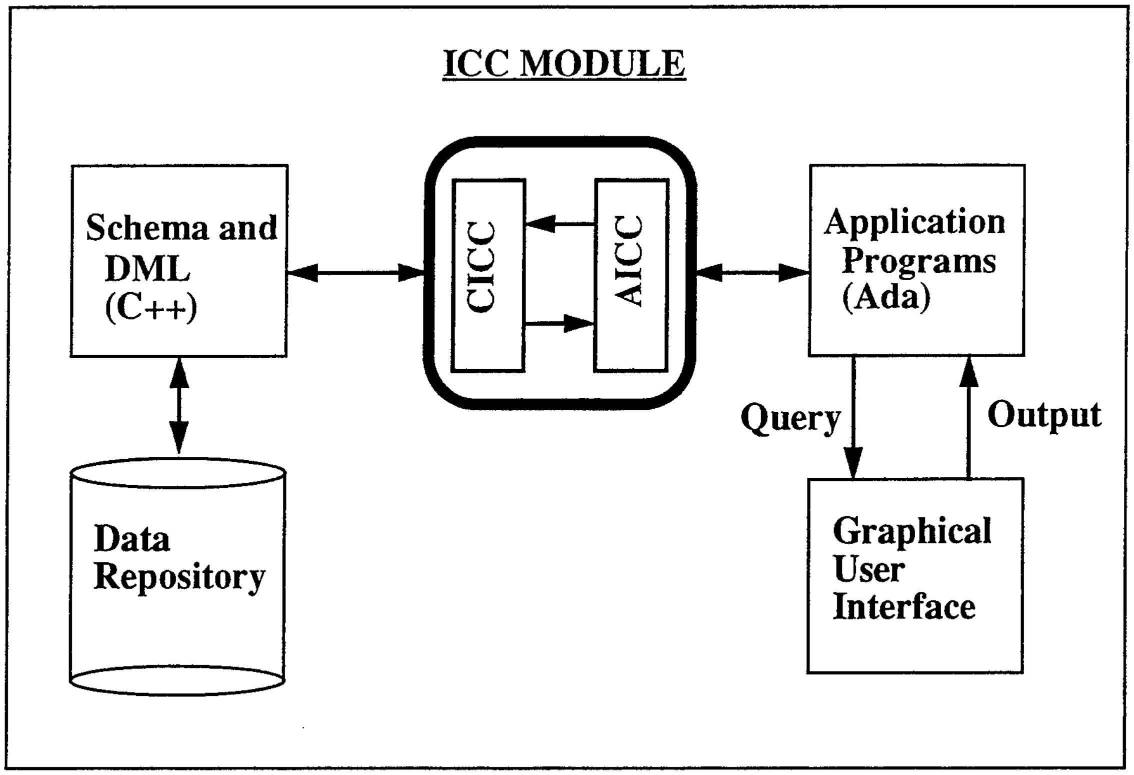


Figure 7.1 The Role of the Inter-Communication Controller

3. Inter-Communication within the ICC

Since our technique relies on the inter-communication between the two parts of the ICC (the AICC and the CICC), a basic requirement is to device a methodology that handles the technical issues for the required communication between them without violating the main objectives of our technique.The methodology we employ is summarized by the following three steps and is clarified by a following example drawn from the experiment template.

1. Compile the C++ code implementing the CICC using a C++ compiler.
2. Use the Unix *nm* command to get the symbolic names of the subroutines comprising the CICC. Choose the symbolic names which are preceded by "*T*".
3. Use the symbolic names obtained in 2 in the "*pragma_interface*" and

pragme_interface_name in the corresponding Ada subroutines.

4. Use *link_with_pragma* to pre-link with the CICC object code in each of the files containing that code (.o files).

Example: Suppose you have an Ada function called *ada_new_designer* and the object code of the corresponding C++ function called *c_new_designer* is in the file *person_inteface.o*. Now you want to interface the Ada function to the corresponding C++ function:

1. Assuming your C++ code is already compiled and you have the (.o) file(s).
2. Use nm:

```
        >nm -ao person_inteface.o | grep "c_new_designer"
```

The output will be a punch of names as follows:

```
person_interface.o:000006c8 T _c_new_designer__FPciT2
person_interface.o:00000000 - 00 0000 LSYM c_new_designer__FPciT2:ZtF
person_interface.o:000006c8 - 00 001c  FUN c_new_designer__FPciT2:F(0,1
```

Choose the first one (preceded by "*T*"):

"*_c_new_designer__FPciT2*" as the symbolic name.

3. In your ada code, all you need is the following:

```
Function ada_new_designer(name : in c_string; level : in integer ; status : in
                                            integer ) return designer;
pragma interface(C,c_new_designer); -- C++ subroutine name
pragma interface_name(c_new_designer, "_c_new_designer__FPciT2");
```

This way you get an Ada function called *ada_new_designer* so when you say somewhere in your Ada code:

ada_new_designer(some_name,some_integer,another_integer)

you are actually calling the corresponding C++ function:

c_new_designer

4. For step #3 you have 3 ways to link with the C++ relevant object files:
   - Use with$_n$ (not recommended by Ada documentations).
   - Supply the C++ (.o) files in the command line as options to a.ld; this is doable and can be included in a Makefile.
   - The way we are using is to use *link_with pragma* to link to the desired (.o) file; for example: *pragma link_with("person_interface.o")* to pre-link with the object file *"person_interface.o"*.

C. DEVELOPMENT TEMPLATE

We developed a template as a guide to be followed that contains the code units required; either C++ or Ada. The structure of each unit and the minimum content is also specified. Some of these units are required and can be used for the rest of application or other similar application and may only need some extension to include other functionalities if the need arises. Some other units are specific to the template case but can be used as templates on other contexts.

One key point about the code units of the template specially those implementing the ICC is that they, except for the C++ implementing the schema, are paired between Ada and C++ in a mirrored image way. Each C++ unit of these has a corresponding Ada unit that has the same attributes and operations. Figure 7.2 illustrates the mirror units in the design database mock up which models a database schema definition and implementation as well as the manipulating application.

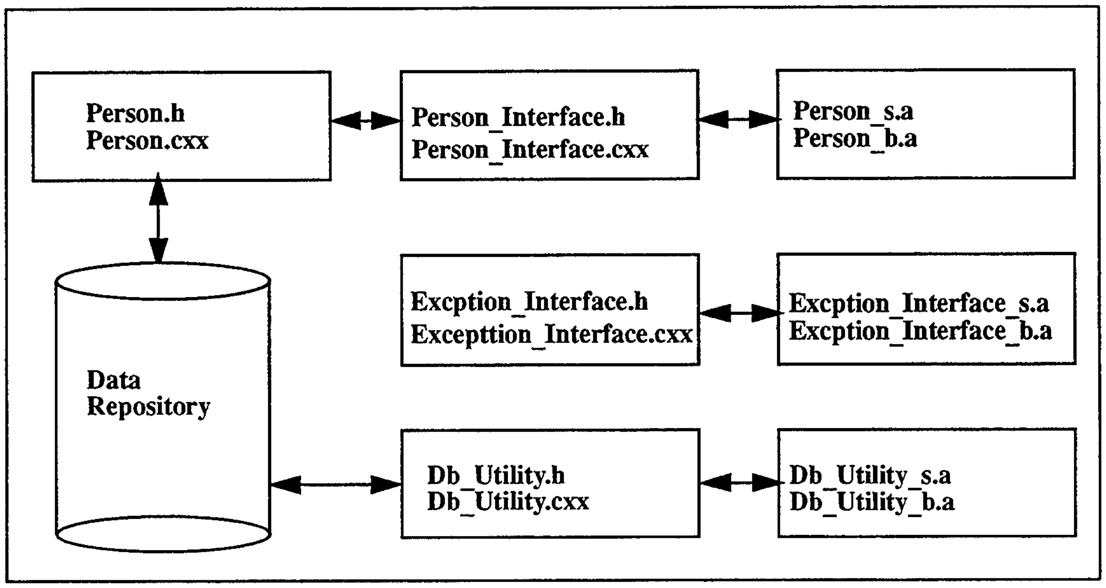


Figure 7.2 Mirrored Units of C++ and Ada

1. Schema Code Units

a. Definition Units:

For each type in the schema there is a code unit representing the definition of the type. This unit and the one that follows are coded using pure C++ code intermixed with the use of ONTOS supplied classes and free functions. Notice that the set of operations given by that unit is the only way to communicate with the unit. Each type unit contains the following:

- Attributes of the type (termed data members in C++ terminology).
- Relationships between the type and other types (reference to other types).
- Primitive operations (methods; termed member functions in C++ terminology) defined over the type (mostly operations to set and get attribute values of objects).
- Operations required by ONTOS especially for persistent objects such as storing and retrieving objects into and from the database. Note also that the

implementation of the primitive operations and relationships normally use advanced constructs beyond the limit of standard C++ and supplied by ONTOS such as aggregates, iterators, storage management, etc.

b. Implementation Units:

For each definition unit in 1. there is another unit that represents the implementation of the definitions given by that unit. These units hides the implementation detail to achieve the object-oriented concepts of encapsulation and information hiding.

2. Supporting Units

The supporting units implement the CICC which is used (with the AICC) as a bridge to communicate the Ada code units and the schema code units. These units are coded like the above ones in pure C++ code intermixed with the use of ONTOS supplied classes and free functions and include the following type of units:

a. Interface Units:

For each definition or implementation unit there is an interface definition or implementation unit. Each pair of the interface definition and implementation units contains only the operations subset of the corresponding definition and implementation unit pair. This subset is redefined and reimplemented as the definition and implementation of a set of free functions (not class methods). Each of these free function has an extra argument which is a pointer to the schema type represented by the class. This extra effort was made to make it possible for Ada to call C++ class methods indirectly, we did not succeed to do without this artificial second level provision. Especially we could not get the symbolic name for the C++ class constructors to be used in the Ada pragmas. Additionally, C++ operations work on the context object; the object for which the function was called. For example when a C++ class function (operation) named *print_data()* is executed, actually there is a hidden (object) argument for this function whose data is to be printed; as if the function reads: *print_data(ObjectX)*. The caller of this function therefore must call it within the context of some object using the dot or arrow notation (*ObjectX.print_data()* if ObjecX is the object itself or *ObjectX=>print_data()* if ObjectX is a pointer to the object). This is easy if the caller is another C++ (or C) fragment of code, but very dangerous and unsafe if the caller is an Ada fragment of code because of the different representations. Even the use of the Ada representation clauses is restricted and unsafe specially with big and complicated data structure like the ones represented by C++ classes. Therefore the only way to allow Ada to call C++ methods was to make it indirectly through the interface units using object pointers as an additional argument to these functions. The called function will then dereference the pointer as the context object and call the corresponding class function. Further the communication between the Ada code and the interface units is through pointers only under conditions that these pointers **must not be dereferenced in the Ada side.** For example to create or retrieve an object, Ada code calls the appropriate free function in the interface unit (say *createX or lookupX*) using a dummy pointer (points to an empty record) as an additional actual parameter. The free function then dereference the pointer as the context object and calls the appropriate class member function and returns a pointer to the object back to Ada. Later, if Ada needs to perform any operation on the returned object, it calls the appropriate free function passing any required parameter in addition to the object pointer that it received earlier. This means that the communication is actually implemented by passing pointers back and forth between Ada code and the interface units. Since Ada is not allowed to dereference the pointers it receives, it would be impossible for Ada to call any class function directly unless this function has an additional parameter for the object it will work on which makes a C++ class something else not related to C++. This was another reason for using a second level interface units.

b. Database Utility Units:

These is a single pair of definition and implementation unit that include a set of free functions which provide Ada with interface to some relevant Ontos DB operation so that they are visible to and callable from Ada. These free functions communicate directly with the ONTOS supplied free functions. With some additional effort these functions can be eliminated totally and let Ada communicates directly with the ONTOS supplied free functions which is relatively easier than communicating directly with pure C++ classes. This can be done by extracting the symbolic names of these functions and using them in the Ada side. However this requires the examination of the ONTOS object files that define and implement these free functions.

This pair is general and can be used in similar application and can be extended in the future in the same way as the need arises to include other database functionalities. Currently it includes the following set of free functions:

- An interface to ontos DB operation that opens the database *OC_open(dbname)*
- An interface to ontos DB operation that closes a previously open database *OC_close(dbname)*
- An interface to ontos DB operation that starts a transaction *OC_transactionStart()*
- An interface to ontos DB operation that commits a transaction *OC_transactionStart()*

c. Exception Interface Units:

These is a single pair of definition and implementation units that include a set of free functions used to capture exceptions raised from inside ONTOS DB and map them to integer indices. These indices are returned (for check) to Ada by a C++ function made visible and callable from inside Ada (exception_index function). In the Ada side, these exception are handled properly in a way that does not allow the program to abort and if an abort is unavoidable, the handler can prompt the user with a meaningful message and possibly a guide as what to do. Depending on what exception has occurred, some actions are performed. The most noticeable of these actions is the case where the exception was raised because of a user error (e.g., a misspelled DB name) in the course of opening the DB operation. In this case the user is prompted to correct and reenter the DB name and the operation is retried again. The exception interface units are general and can be used in any similar application even under different context. Some of the codes and their meaning for the ONTOS captured exceptions to be handled in the Ada side are as follows:

- Normal_code: no exception has occurred.
- Object_already_exists_code: an attempt was made to store object into the DB but another object is already exists in DB having the same name.
- No_such_object_code: no such object in the DB having that name.
- DB_open_failed_code: an attempt was made to open a DB that does not exist or using a wrong name (e.g., misspelled).
- DB_not_open_code: an attempt was made to close a DB that was not previously opened.
- No_active_transaction_code: an attempt was made to commit a transaction that was not started yet.
- ONTOS_Failure: a catch all that capture any other exception from ONTOS not covered by the cases above.

3. Ada Images of the C++ Code Units

The Ada code structure comprising the template and the application in general has two main parts: the first part represents an approximate mirrored image for the C++ code units in *a*, *b*, and c above, and the second part represents the user interface. Each Ada unit is represented by an Ada package with a specification and a body. The specification part represents the public interface to the unit. The body includes implementation details which are not required to be visible outside the unit. In the following we discuss the structure of the Ada code that belongs mostly to the AICC part.

a. The Schema Types Images:

Each of these Ada units has a specification and a body corresponding to the definition and implementation units of the C++ realization of the type. Each paired specification and body of these units models an Abstract Data Type (ADT) for each schema type. The user sees only the visible operations and exceptions of these ADTs and is not aware of how the communication to the corresponding lower level C++ code is implemented. These implementation details are hidden in the body unit (package) of each pair. Each specification contains the same visible operations given by the C++ definition of the corresponding schema interface unit. Naming convention is followed so that the C++ name of the operation is augmented with the prefix "c_" which is removed from the corresponding Ada name. Each body mainly contains the pragma portions, an exception handler, and any additional Ada logic. The pragma portions implement the communication for each operation to the corresponding C++ one defined and implemented in the schema interface unit. The pragma portions for each operation include pragma_interface, pragma_interface_name, and pragma_link_with discussed in section B.3. An example of the use of these pragmas to interface an Ada function to a corresponding C++ is shown in Figure 7.3. The exception handler handles any possible captured exception that may be raised by ONTOS including the catch-all case.

```
Function ada_c_GetPersonStatus(d : in designer)
            return integer;

 pragma interface(C,ada_c_GetPersonStatus);

 pragma interface_name(ada_c_GetPersonStatus,
            "_ada_c_GetPersonStatus__FP13Person_ENTITY");
```

Figure 7.3 Pragma Usage

b. The Database Utility Image:

This Ada image includes one specification and one implementation (body) unit each corresponds to the definition and implementation of the imaged C++ and ONTOS one and includes the same operations. This utility allow Ada to directly invoke ONTOS DB general operations to open and close the database, start and commit a transaction, and to save and delete objects to or from the database.

This pair is general and can be applied to other applications. It can also be extended in a similar way to include other database functionalities. Similar to the image in 1. above, the body hides the detail of the actual communication implementation and also includes the same main parts. Fragments of the specification and implementation parts of this unit is shown in Figure 7.4 and Figure 7.5 respectively. Notice the use of exceptions in the implementation fragment as will be explained next.

```
package db_utility_PKG is

-- General ONTOS operations

   procedure open_database(ddb : in a_string);

   procedure close_database(ddb : in a_string);

   procedure transaction_start;

   procedure transaction_commit;

   procedure save_to_db(d  : in designer);

   procedure delete_from_db(d : in designer);

end db_utility_PKG;
```

Figure 7.4 Specification Fragments of the DB Utility Package

```
procedure open_database(ddb : in a_string) is

  begin

  ada_c_open_database(to_c(ddb));

     case get_exception_code is
            when normal_code                  =>
                      null;
            when db_open_failed_code          =>
                      raise db_open_failed;
            when others                       =>
                      raise Ontos_failure;
     end case;


  end open_database;
```

Figure 7.5 An Implementation Fragment of the DB Utility Package

c. The Exception Interface Image:

This Ada package specification and body represent a mirrored image for the corresponding definition and implementation of the C++ exception_interface units. It is made in a separate package because most of these exceptions apply to all types. Some of the exceptions can also be renamed to suit a specific type. The syntactic use of these exceptions is made uniform whereever needed. The defended operation is performed first and then a check is made for the occurrence of any possible exception including any unexpected one by checking the occurrence of the catch-all case. If no exception occurs as a result of performing the operation, the normal path is completed. Otherwise the corresponding exception is raised and the processing starts at the appropriate exception handler where the proper action is taken.

For example, to perform an operation called *find_designer* on the schema type *designer* that retrieves a designer from the database given his name, the syntax of the operation (an Ada function in this case) is illustrated in Figure 7.6 below.

```
begin --find_designer
   d := ada_c_find_designer(designer_name);
   case get_exception_code is
      when normal_code                  =>
             null;
      when no_such_object_code          =>
             raise no_such_designer;
      when others                       =>
             raise ONTOS_Failure;
   end case;
   return d ;
end find_designer;
```

Figure 7.6 The Syntax of Exceptions Usage

Where `ada_c_find_designer` is the corresponding C++ function and `get_exception_code` is the exception interface function that returns the exception code. If no exception occurred, the value of the designer held by d is returned, otherwise the exception is raised. In the context of this operation, the only exception that may be raised is the "`no_such_object`" one which means that a designer with the given name does not exist in the database. However to guard against any unexpected exception from ONTOS side, the catch-all exception "`ONTOS_Failure`" is used.

d. Iterators

As part of our template, we have developed an Ada binding for ONTOS iterators. An iterator is one category of operations that can be applied to an object to visit all parts of that objects [26]. Depending on how much abstraction is exposed to the outside view, iterators are classified into either active or passive. In the active approach, an iterator is exposed as a set of primitive operations opposed to a single operation in the case of the passive approach. According to Booch [26], active iterators give great flexibility and can be easily used in composing reusable software components that build on top of an iterator. We followed the active approach in implementing iterators to iterate through the structures of our schema objects. One important class of these structures that normally require the availability of iterators is the aggregates class embedded in other objects, these include lists, sets, maps, etc. Ada provides iterator mechanisms for simple types only like arrays using e.g., a *for loop.* We developed the iterator template for complex objects using the basis provided by ONTOS. In our approach, an iterator is considered an object of an abstract data type that has the following operations:

- `CREATE_ITERATOR`: creates an iterator of the proper type.
- `Has_More_Elements`: returns True if the iterator has visited every item
- `Get_Next_Element`: returns the next item in the iteration.
- `RESET_ITERATOR`: resets the iterator causing it to start iteration again from the beginning using either the same object or new one of the same type
- `Destroy_Iterator`: to decollate memory occupied by the iterator

The syntax of using the iterator is shown in Figure 7.7.

```
Designer_Iter := CREATE_ITERATOR ( type_name );

While Has_More_Elements(Designer_Iter) loop

 Designer_Object := Get_Next_Element(Designer_Iter);

 do something with Designer_Object;
          .....
          ....
end loop;
```

Figure 7.7 Iterator Usage

D. AN EXTENSION TO THE TEMPLATE

The types of our conceptual model are linked together by complex associations. Many of these associations are time varying too. They are all binary relations. The majority are many-to-many and some are one-to-many or many-to-one. One characteristic of the instances of these associations (at least the ones we are considering) is that they are sets of objects. Any implementation of our model and the decision support mechanism should consider an efficient representation and implementation of these relationships. This consideration should:

1. Provide a container for the set of objects of the association instance.
2. Build instances consistently in both directions of the association (domain and codomain).
3. Provide a generic representation and implementation for all these association.This is accomplished by having a single generic enough template that can be used as a derivation standard for all associations.

The limitation of the ONTOS DB, the implementation database engine, restricts the realization of the above objectives. Even with the new releases of ONTOS, the generic scale we are willing to represent can not be provided. Refer to C.2 of ChapterVI for details.

To achieve the above objectives, given ONTOS restrictions, we extended the basic implementation template we developed in this chapter. This extension is summarized in the following:

1. The design and development of a binary relation type which we included as one of the schema types. Objects of this type can capture instances of all binary relations addressed by our model.
2. The implementation of the binary relation type is made generic enough to be used by all types with the proper meaningful naming.
3. Macros was used in the implementation of the binary relation type to provide a generic representation for functionality and naming.

4. Instances of this type are generated automatically using macros.
5. The binary relation type is inherited by all other types that have such associations.
6. Each inheriting type is provided (using macros) by two instance for each embedded binary relation. One in the direction from the domain to the codomain and the other in the opposite direction.
7. Operations provided by an instance of the binary relation type provides all basic needs of the relation's embedding type in terms of operations to be performed on the association.

This extension of the basic template has the advantages of consistently establishing associations, controlling their both directions, reduce coding effort, simplify code design, and improve maintainability of the code.

Appendix B includes the code for samples of a type implemented using the binary relation type. It also includes the different levels of macros definitions and instantiations used. This experiment needs more examination and refinement that we could not do because of the time and some problems related to the version of ONTOS we are using.

VIII. CASE STUDY

The case study we are about to deal with is a prototype for a generic *Command, Control, Communication,* and *Intelligence* (*C^3I*) *System.* This system was developed using the CAPS design environment [84]. C^3I system as was developed in CAPS can be implemented in wide variety platforms in support of a Composite Warfare Commander (CWC) command and control architecture [68]. The system forms a network of (possibly LANs of) generic C^3I stations. Each station is a specialized instance of a common design. The architecture provides for connectivity between naval platforms, shore-bases, and external forces. It enables the processing of tactical data from internal and external sources. The workstation provides the CWC, his subordinate commanders, and coordinators with a system that supports them in monitoring air, surface, subsurface, and power-projection (strike) tactical environments.

C^3I systems are characterized by an inherent complexity of both requirements and design. Being a typical of real time embedded systems makes it more complex. It includes distributed processing, hard real-time constraints, and multiple hardware interfaces [46].

For the case study to be focused given the above characteristics, we will not cover all aspects of such systems. Instead, we concentrate on a subset of the system requirements at a given point of time during the design and development process. We then show how such a requirements subset changes as a result of the criticisms received from stakeholders in response to the demonstrated behavior of the prototype at the given state. We also show how stakeholders participate in refining and elaborating requirements supported by the automated aids based on a formal model. The automated process for propagating the consequences of the requirements changes down to the design hierarchy is also shown. For the detailed requirements of such a system refer to [68]. For the design and implementation of the prototype reflecting these requirements refer to [84].

A schematic diagram of a generic C^3I station is shown in Figure 8.1. The figure shows the external systems with which the proposed system communicates. These include the users, weapon systems, platform sensors, navigation system, and communication links.

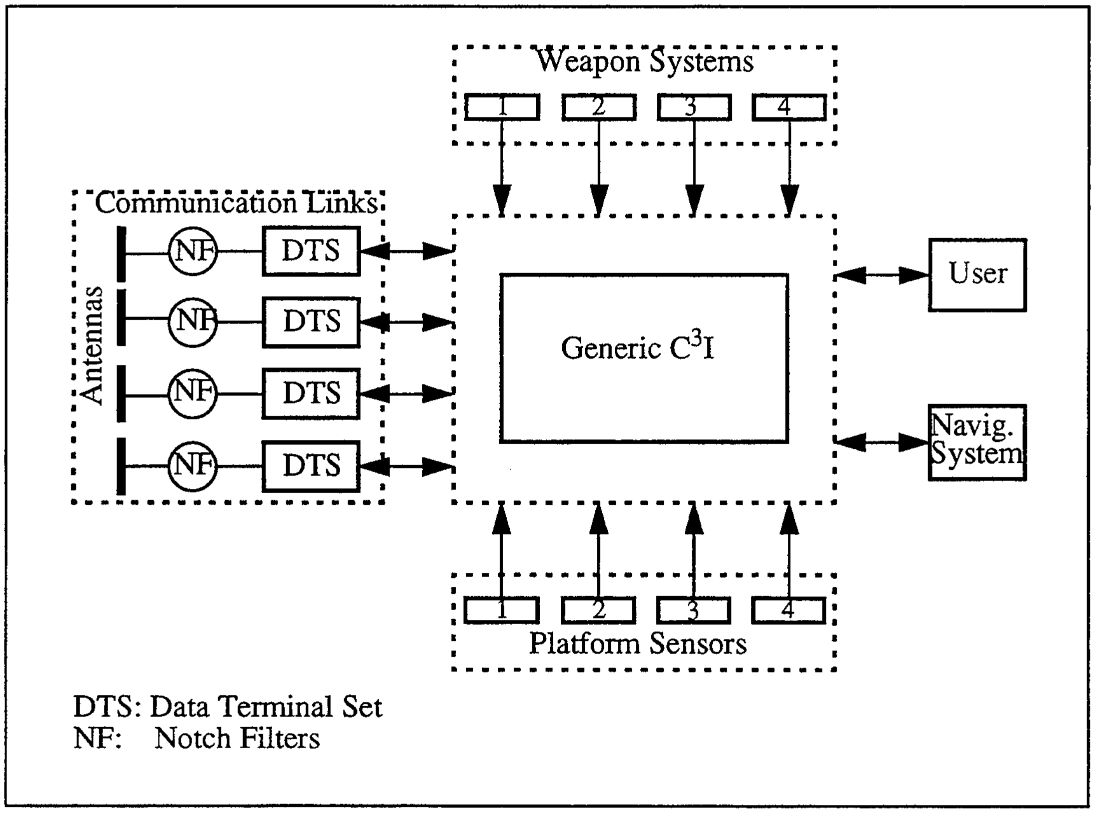


Figure 8.1 External Interfaces of the Generic C^3I System [46]

Following is a brief description of external interfaces [68]:

1. User: could be a CWC, officer in Tactical command, Warfare Area Commander, Tactical Action Officer, Communication Officer, etc.
2. Communication Links: any digital communication system capable of transmitting and receiving digital messages.

3. Platform Sensors: any locally-mounted device capable of identifying azimuth, elevation, velocity, and/or heading of a contact or track is considered to be a platform sensor.
4. Navigation System: a system that provides a platform with own positioning, course, velocity, and time data.
5. Weapons Systems: this interface, if exists, makes the weapons status information available to the battle manager.

The information flow into and out of the proposed generic C^3I station is shown in the context diagram given as Figure 8.2.

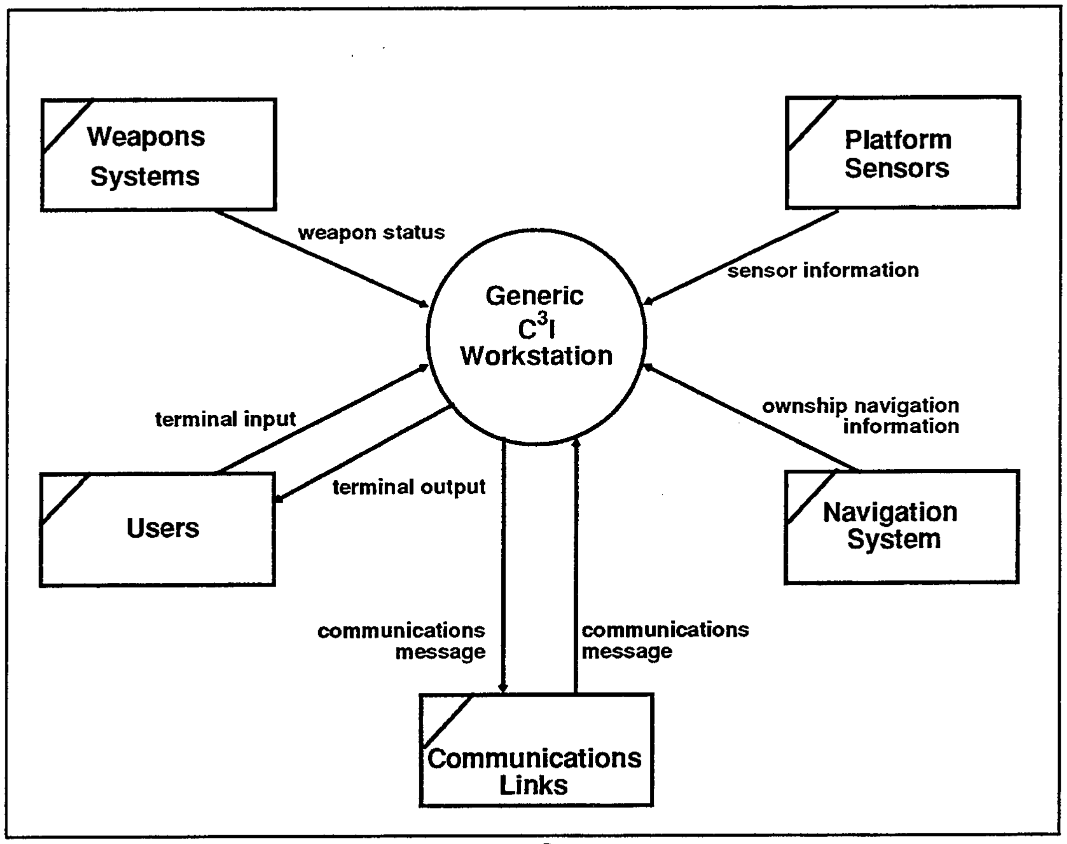


Figure 8.2 The Generic C^3I Context Diagram [68]

A. THE REQUIREMENTS SET

As we said earlier, we concentrate on only a subset of the system requirements and the corresponding system design components. This subset includes the requirements related to the track database (TDB). This database is the reservoir where tracks and their related information are stored.

1. Current State of the Requirement Components

The current state of the requirements hierarchy structure includes the following components. The notation Ri.j is used to denote the hierarchical structure of the requirement components. It means the *j*th child of the requirement component *i*.

R1 The generic C^3I must provide a TDB capable of efficiently storing, accessing, and updating track information in real time.

R1.1 The TDB must provide for storing tracks received through all sources.

R1.1.1 The TDB must provide for storing tracks received through communication links.

R1.1.2 The TDB must provide for storing tracks received through platform sensors

R1.1.3 The TDB must provide for storing tracks entered manually by the user.

R1.1.4 The time for storing a track into the TDB must not exceed 1 second

R1.2 The TDB must allow the user to delete tracks.

R1.3 The TDB must allow the user to change tracks.

R1.4 The TDB must allow the user to retrieve tracks.

R1.4.1 The response time for retrieving a 1000 track information must not exceed 1 second.

These are the requirement components identified so far that relate to the TDB. Figure 8.3 shows the hierarchy of these components. This hierarchy is represented by *PartOf* relationship.

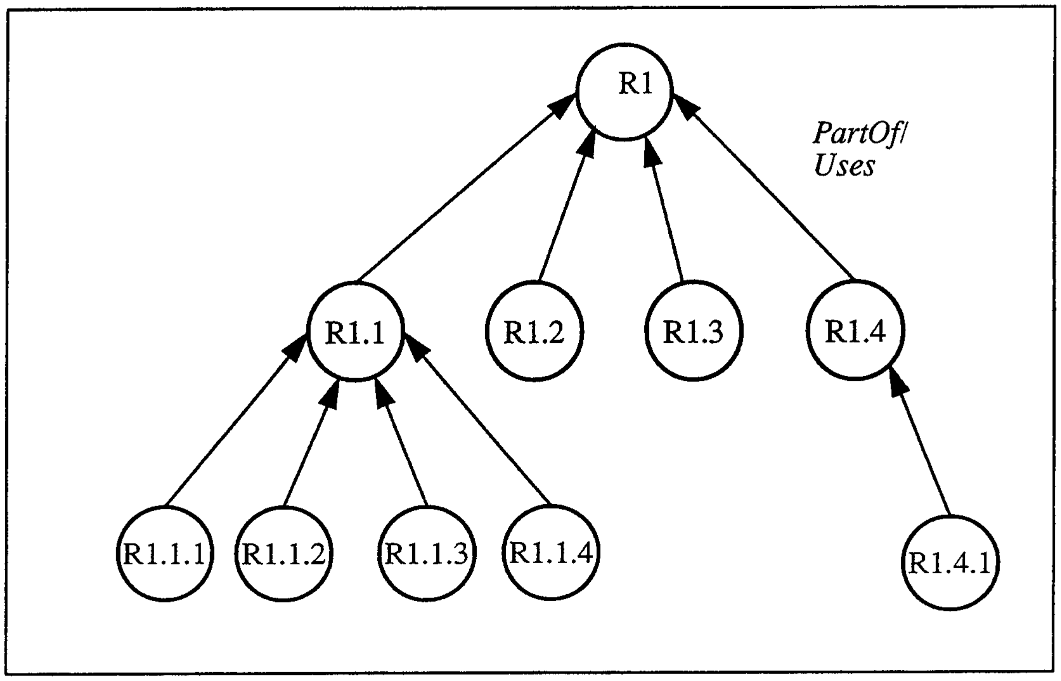


Figure 8.3 Requirements Hierarchy

B. CURRENT STATE OF THE DESIGN

To see where the TDB subsystem is with respect to other major subsystems of the generic C^3I system, Figure 8.4 gives a first level module decomposition of that system. The current state of the design that maps the TDB requirements subset is given by the corresponding fragments of the PSDL flow diagrams shown in Figure 8.5, Figure 8.6, and Figure 8.7.

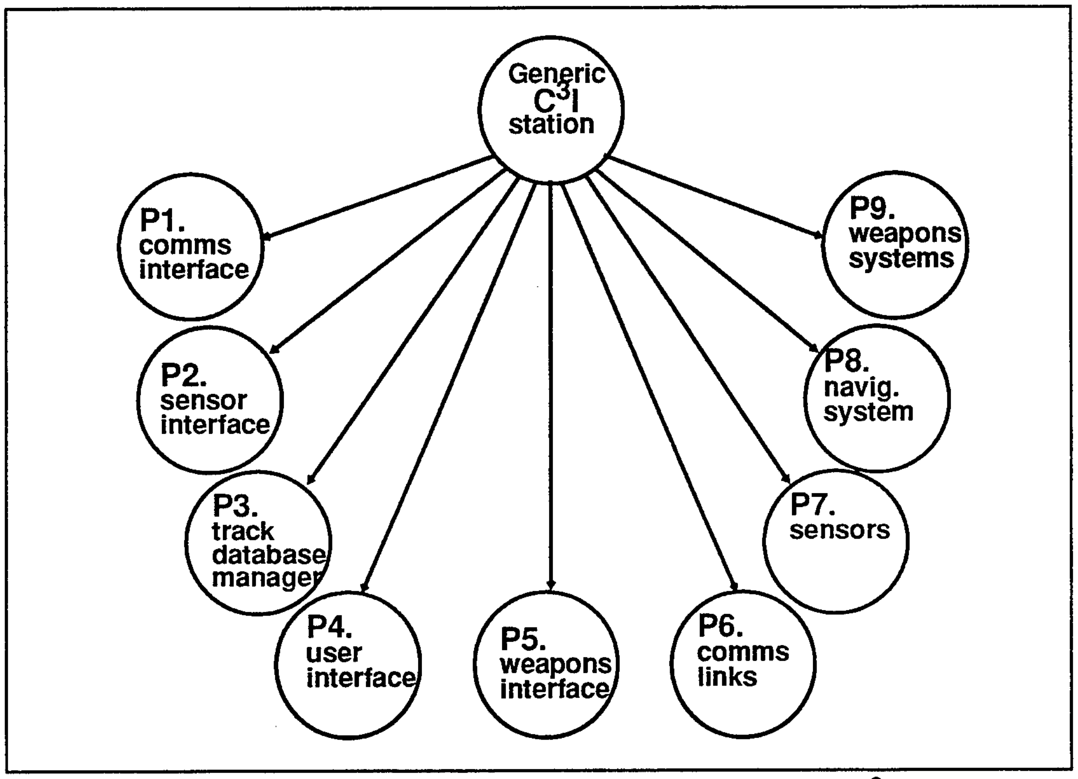


Figure 8.4 Module Decomposition of the Generic C^3I

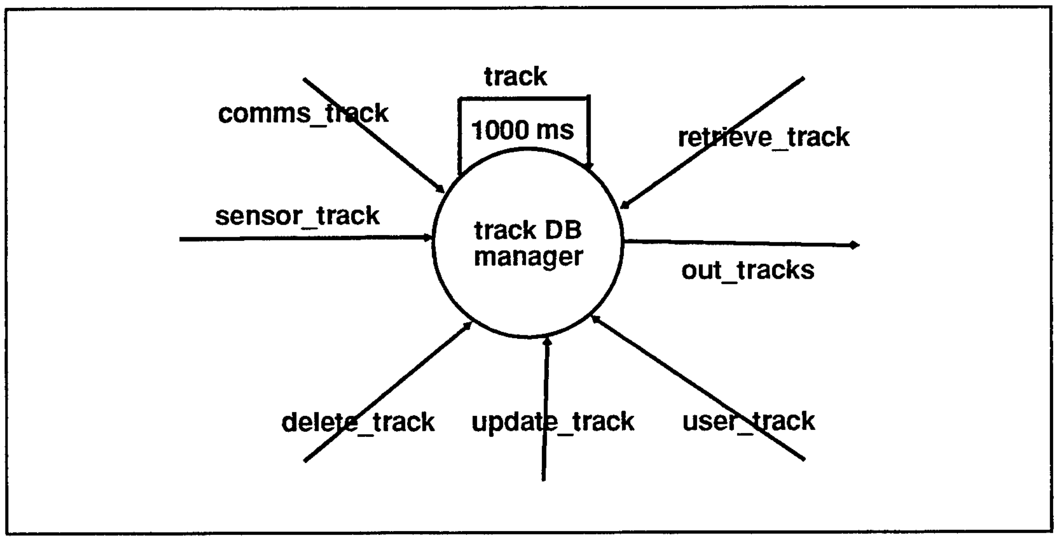


Figure 8.5 The Track DB Manager Module

In Figure 8.5 the TDB function is represented by one (composite) PSDL operator. This operator models a state machine whose state variable is the "track". The figure also shows the input and output data streams to and from the TDB manager operator. This operator has a maximum execution time (MET) of 1000 ms. For details about the allocation of this or any other time constraint either for this operator or any other operator that follows, refer to [84].

Figure 8.6 provides the first level decomposition of the TDB manager operator. The decomposition includes PSDL operators for adding, deleting, changing, and retrieving tracks from the database.

The operator "add the track" is further decomposed as Figure 8.7 shows. This is to account for the addition of tracks from three different sources: communication links, platform sensors, and user tracks.

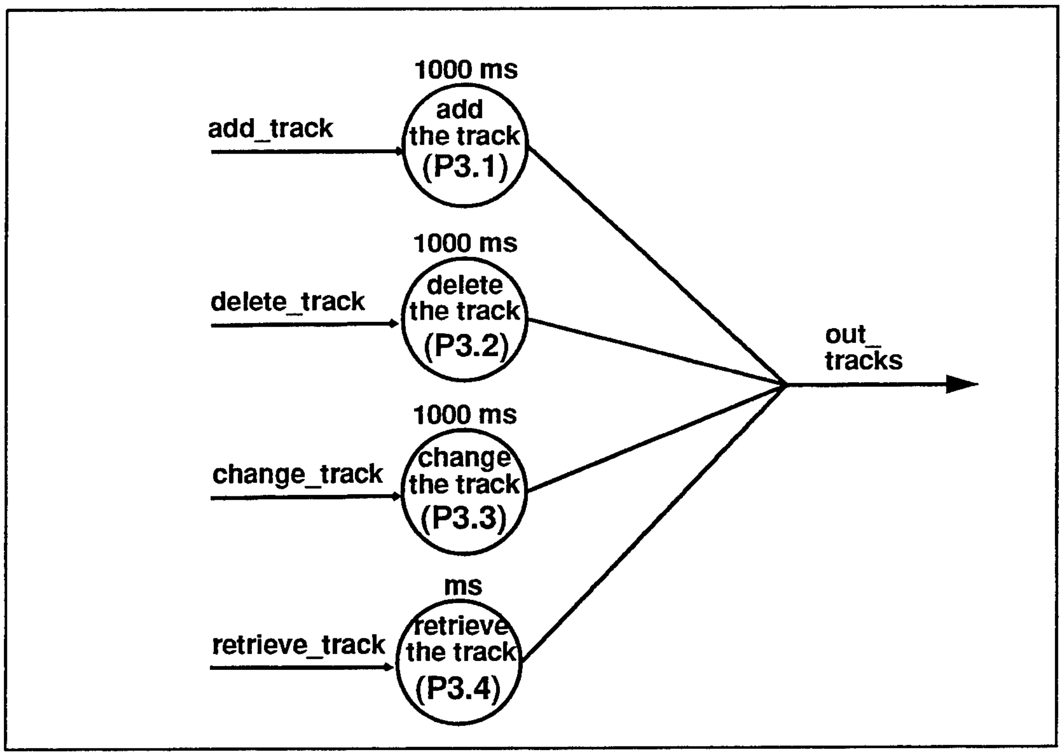


Figure 8.6 First Level Decomposition of the TDBM

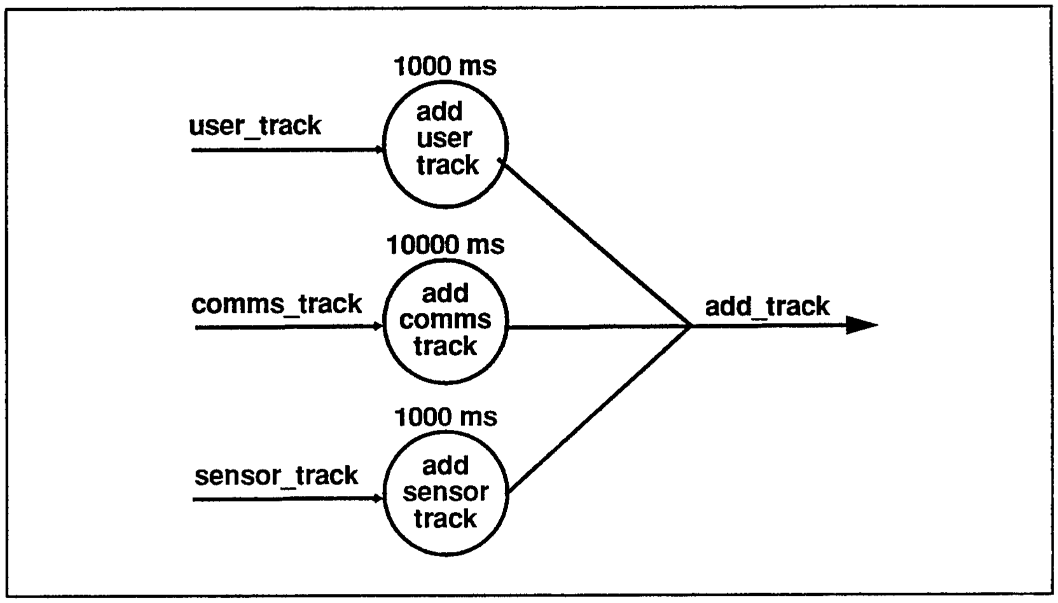


Figure 8.7 Decomposition of the Operator "add the track"

C. DEMONSTRATION

At the target date, the behavior of the system prototype is demonstrated to the stakeholders in the presence of the design team members. The stakeholders list includes 5 persons representing different groups of the potential users. The design team is represented by four persons.

1. Individuals

Individuals from the customer side include a representative for each of the following stakeholders (SHs). From now on we refer to them as SH1, SH2, SH3, SH4, and SH5 respectively.

1. Composite Warfare Commander.
2. Strike Warfare Commander.
3. Force Coordinator.
4. Track Controller.

5. Communication Officer.

Individuals from the design team include:

1. The manager.
2. An analyst.
3. Two designers.

An object that represents each of these individuals is already in the design database. The attributes of each such objects is assigned the pertinent values corresponding to each individual. These values carry the names, organization, roles, expertise, etc. Refer to section C.3 of Chapter VI for the complete attributes.

2. Criticisms

Using the system, the analyst creates a demonstration step. This step is then linked to the objects of the present individuals, the current version of the prototype, the set of criticisms from the last demonstration, and any scenarios set for the demonstration. The set of criticisms are the primary input to the step. The rest are secondary inputs. The analyst then starts the demonstration by issuing *execute* command. In response to the demonstration, the following criticisms are posed by three SHs:

1. **Criticism1**: *There is no discrimination in the track database between new and old tracks.*
2. **Criticism2**: *When reporting the same track, track reports received from different external sources are misleading.*
3. **Criticism3**: *The displayed tracks, when requested, must express the focus of the requester.*

The analyst creates an object for each of these criticisms, links it to the person who raised it. This object is then entered to the design database. The analyst and designers are then engaged in a conversation with the SHs to elaborate on and analyze these criticisms to better understand the user requirements. The result of this conversation for the meaning of the first criticism was that with the anticipated high rate of tracks arrival from different sources, keeping them all in the same database slows down the retrieval process. It also causes the display of obsolete tracks.

For the second criticism the conclusion is that with the potential of the track sources being remotely located, the same track reported can enter the database more than once. This is due to the communication delay. For example assume that a track T is reported at the same time point by two different sources S1 and S2. S2 is remotely located. The report message from S1 arrives before that of S2 and both are entered into the TDB as two different tracks.

Exploring the last criticism reveals that the system should allow the user to specify retrieval and display criteria of the tracks when requested. When the user is interested in aerial tracks, only aerial tracks are retrieved and displayed. This reduces the number of tracks retrieved and displayed, and serves the focus of the user. This focus can be distracted by too many tracks displayed while only a subset of them is currently needed.

The analyst attaches these elaborations to the criticism objects he created. He also compares the criticism to any related ones from the last demonstration to look for conflict in interests. He then links these refined and elaborated criticisms to the analysis step as its output. The detail regarding the transition of the step from the proposed state until it outputs the refined criticisms and completes remains the same as was explained in Chapter III.

3. Issues

The completion of the demonstration step automatically triggers the creation of another step: the issue analysis step. The system assigns the issues from the last demonstration to the newly created step as its primary input and the refined set of criticisms currently generated as its secondary inputs.

The next task the analyst does is to acquire and synthesize a set of issues from the criticisms of the SHs. He does this using also the process history. The history for this task is approximated by the set of issues from the last demonstration. If one issue can be the generalization or the specialization of a previous one, this makes the resolution process much more easier by adapting the artifacts used to resolve it. The new issue may also has its solution in one of the abandoned alternatives for the old one. The new issue may contradict the resolution of an old one. These are some of the reasons for recording the history of the process; design rationale is not lost.

The issues synthesized from the refined set of criticisms are:

1. **Issue1**: *The undistinguished coexistence of current and obsolete tracks in the TDB is not recommended.*
2. **Issue2**: *The TDB must not allow the storage of duplicate tracks received from different sources (including local sensors).*
3. **Issue3**: T*he TDB must allow the user to filter the retrieval and display according to his focus.*

In this simplified case, it happened that the mapping is one-one between the criticisms and the issues. This is not the case in general though. One criticism may map to two or more issues and vice versa, i.e the mapping is many-to-many in the general case.

The analyst reviews the currently synthesized issues, compares them against the previous ones, and creates an object for each using the system. These objects are then linked to the step as its output. The analyst also establishes the link manually between the issues and the criticisms from which they were synthesized. At this point the portion of the new state of the dependency graph acquired so far is given in Figure 8.8. The design database stores the same image for this portion too.

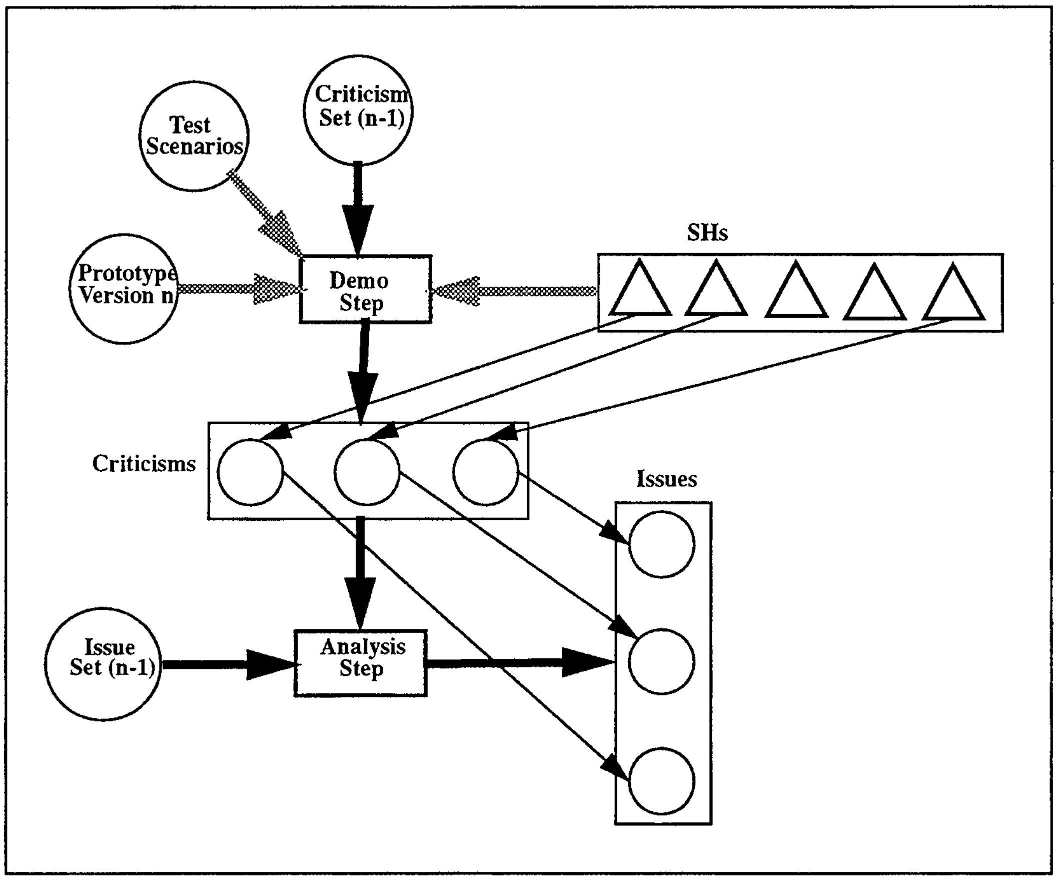


Figure 8.8 Part of the Dependency Graph at the End of the Issue Analysis Step

4. Issues Resolution

Resolution of each of the acquired issues requires: first, deciding on the available alternatives for the resolution. Second, determining the requirement components in the existing requirements hierarchy affected by each alternative. Third, Proposing new requirement components, link them to the requirement hierarchy, and link them to both the issue and the alternative with the appropriate link. Following that is the formal deliberation according to the approach we developed in Chapter V. to reach a final decision for resolving the issue. An automatically generated plan is identified for carrying out the consequence of the resolution on the affected parts of the system design. For each issue, all the above tasks (except for the last one) are performed within the context of an analysis substep as we explained in Chapter III.

In the following subsection we discuss the resolution of each of the three identified issues. For the second and third issues we only provide the conclusions of the corresponding resolutions. For the first issue we provide the detail of the whole process including the formal deliberation to reach a group decision. This is because the process is the same for all issues. Doing this keeps the discussion focused while provides the means for illustrating our approach. The formal deliberation process is given in a separate section.

a. The First Issue

The resolution of this issue will be covered in detail in the next section.

b. The Second Issue

The discussion on the second issue isolated the reasons for the problem in the communication delay for tracks received from remotely located sources. The same problem can occur if a track is received over a congested communication link or due to any other network problem even if the sending station is not remotely located. The suggested solution to this problem was to *time stamp* this latter tracks. Each track received over a communication link is stamped with its arrival time at the destination. The stamp carries the local time of the site. This applies only to the copy of the track to be stored in the site TDB. The copy of the same track to be relayed to other sites will not be stamped.

This way a chronological order can be established between tracks that have the same track ID. It is up to the local focus of the site as to decide on storing policy of such tracks. You should notice that this is related to the first issue. For this reason the resolution of the first issue should come before the second one. This is part of the analyst job; to determine the resolution precedence between issues. The analyst in many cases can determine the interdependencies among issues assisted by the system. This is made possible by using the so far established part of the dependency graph. If the intersection of the affected sets of requirement components of two issue is not empty, then the resolution of one issue precedes the other.

The resolution of this issue modifies the requirement components R1.1.1 which states that *the TDB must provide for storing tracks received through communication links*. The change request (CR) resulting from the resolution is to establish a new requirement component (as a subcomponent of the latter) stating that *tracks received through communication links must be time stamped with the current local time before being stored into the TDB*. Since the requirement component R1.1.1 is linked in the dependency graph to the PSDL component implementing the addition of new communication track to the database, a design change step is automatically generated in the proposed state to carry out the required design change.

c. The Third Issue

The resolution of the last issue was relatively easier. There was no disagreement on allowing the user to filter the retrieval and display of requested tracks. Apparently it is a missing requirement. The resolution of this issue was by modifying two requirement components: R1.4 where the retrieval functionality comes from, and another component that states the display requirement. The latter requirement component is not a part of TDBM requirements hierarchy. The proposed modification is to allow the user, when he requests a retrieval or display, to filter either according to his focus. The analyst, through a discussion with the SHs elicited the meaning of *focus*. According to this elicitation the following elaboration was reached:

1. The user must be able to filter the retrieval and display by the track type (e.g., surface, ground, aerial).
2. The user must be able to filter the retrieval and display by the track range.

3. The user must be able to filter the retrieval and display by the track time.

4. The user must be able to filter the retrieval and display by the track IFF class (friend, foe, neutral).

The proposed modification in requirements affects two design modules: the module responsible for retrieving tracks and the one responsible for displaying tracks. The latter module is outside the TDBM subsystem. It is part of the user interface subsystem. The resolution was to add an PSDL operator to the user interface subsystem to allow the user to enter filtering information. This operator (module) validates this information and sends it to both the retrieval and display operators. This means that within the TDBM only the *retrieve* operator is affected.

D. DETAILED STUDY

In this section we provide a detailed elaboration on the resolution of the first issue. Through this elaboration we demonstrate the establishment of the relationships and inference, alternative exploration and evaluation, SHs' formal debate to reach a decision, the final group decision and how it maps to a change request, the exposition of the proposed change effect on the other parts of the system, and the automatically generated plan to carry out the design and implementation of the change request.

1. Available Alternatives

The issue concern was the separation in storage between current and obsolete tracks. The argument about this concern is that the existence of both, without at least being identified does not serve the focus of the unit. The storage, access, and display of too many tracks while only a subset is useful degrade performance and distract the decision maker.

The discussion revealed that two alternatives are available to resolve the issue:

a. Alternative 1

The first available alternative is to *archive obsolete tracks on an external storage medium.* This alternative is supported by the following arguments:

1. The coexistence of both obsolete and current tracks in the TDB slows down the database access operations.
2. If obsolete tracks are needed, external storage medium can be mounted.
3. Access to obsolete tracks normally occurs under more relaxed situations. Therefore access speed is not an important factor.
4. It is necessary to remove obsolete tracks from the TDB. Otherwise the TDB eventually is filled up.

The primary analysis of this option shows that this alternative includes the addition of the following requirements. We use the notation $(Ri.j)_k$ to mean that Ri.j is one of the requirement components affected or proposed by the $k^{\underline{th}}$ alternative.

$(R1.5)_1$ Obsolete tracks must be archived on an external storage medium.

$(R1.5.1)_1$ The TDB must allow the user to specify current and/or obsolete tracks by date and time.

$(R1.5.2)_1$ Obsolete tracks must be periodically downloaded from TDB into the external storage medium.

$(R1.5.2.1)_1$ The TDB must be scanned for obsolete tracks every one minute.

$(R1.5.2.2)_1$The TDB must allow the user to specify a scan frequency less than one per minute.

The analyst tries to map these requirements into the current design. He concluded that the following modules have to be changed or newly added:

1. Change the user_interface (P4) module in the following way:

- Add a new submodule (P4.6) that accepts and validates the user input to specify what current and obsolete tracks are. Notice that this and the following module are outside the TDBM.
- Change manage_user_interface (P4.1) submodule so that the above functionality is added to the selection menu.

2. Add a new module, monitor_tracks (P3.5), that has the following two submodules:
   - A submodule (P3.5.1) that checks the TDB every minute, removes obsolete tracks and archives them into the external medium.
   - A submodule (P3.5.2) that periodically checks the database to update the status of tracks (tracks_status) and mark tracks that become obsolete since the last update.

b. *Alternative 2*

The second available alternative is to *properly identify obsolete and current tracks in the TDB. Make the default access to current tracks unless otherwise specified by the user.*

This alternative is supported by the following arguments:

1. *Obsolete* and *Current* are relative: what is seen as obsolete by some units under some situation may not be seen so by other units. So both obsolete and current tracks need to stay in the database.
2. The speed of access to obsolete tracks may sometimes have the same importance as that of current tracks.
3. Cost of additional hardware.
4. Adding new hardware for archiving increases the system size and weight which is not suitable for some platforms like airplanes.

The primary analysis of this option shows that this alternative includes the addition of the following requirements:

$(R1.5)_2$The TDB must provide the storage for both obsolete and current tracks, and allow the user to retrieve either or both with the same efficiency.

$(R1.5.1)_2$The TDB must allow the user specify current and/or obsolete tracks by date and time.

$(R1.5.2)_2$ Obsolete and current tracks must be identified in the TDB.

In mapping these requirements into the current design the analyst reached to the conclusion that the following modules have to be changed:

1. Retrieve Tracks (P3.4): must be changed to quantify the retrieval by current, obsolete, or both.
2. Display Tracks(P4.7): must be changed for the same reason in 1. Notice that this module is outside the TDBM.

The analyst also concluded that the following modules have to be added:

1. A module (P4.6) that accepts and validates the user input to specify what current and obsolete tracks are. Notice that this module is outside the TDBM.
2. A module (P3.5) that periodically checks the database to update the status of tracks and mark tracks that become obsolete since the last update.

c. Modifying the Dependency Graph

At this point the analyst does the following:

1. Creates two alternative objects, one for each available alternative, and assigns the value *tentative* to its status attribute.
2. Links each alternative object to issue1 by *MayResolve* link. The opposite direction of this relationship (*MayResolvedBy*) is established automatically.
3. Creates a requirement object for each newly proposed requirement component.
4. Adds each newly created requirement object to the alternative it belongs to.
5. Links the newly created requirement objects in each alternative by the proper *PartOf* link.

6. Links each newly created requirement object to its proposed parent requirement object (if any) in the existing requirements hierarchy.

The newly created objects and links are shown in Figure 8.9. R1 is the root of the TDB requirements hierarchy. Both alternatives have the requirement components set as shown in the figure.

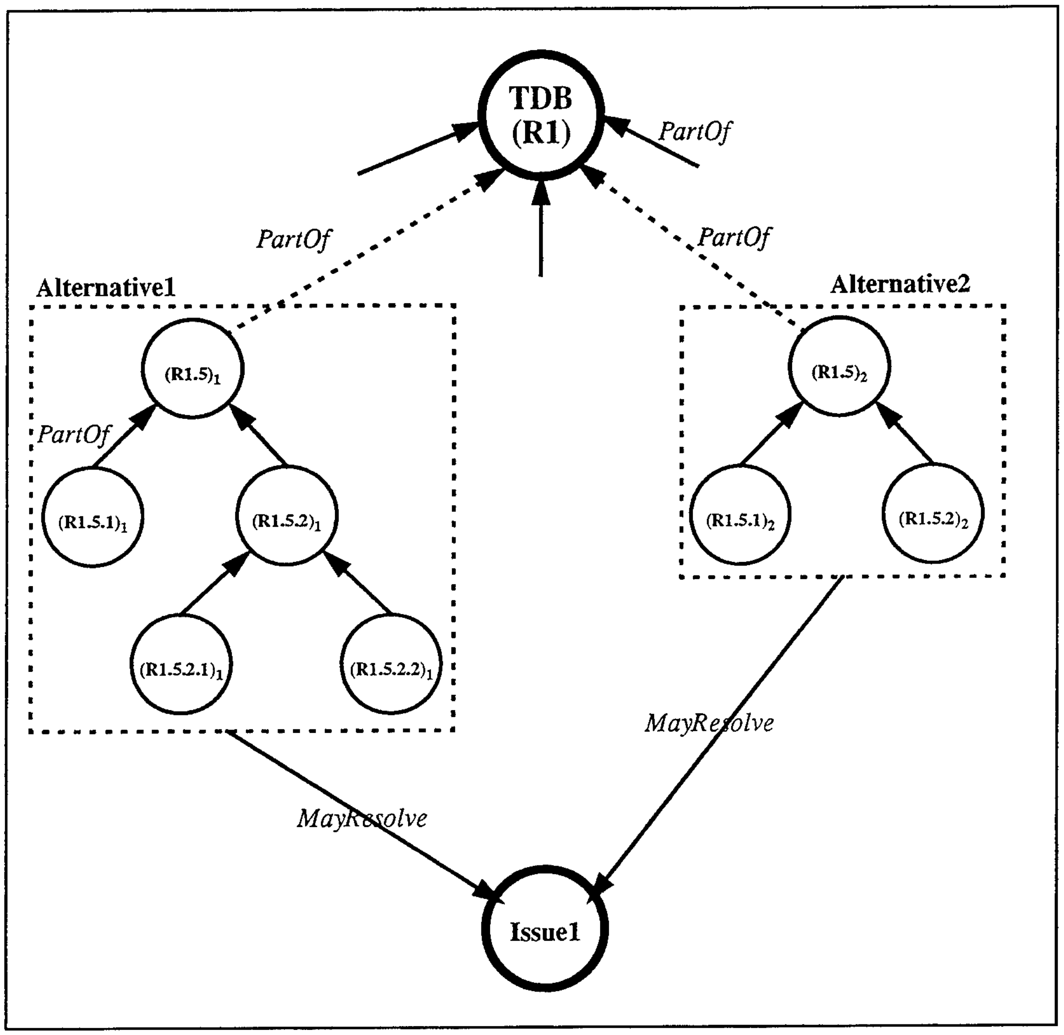


Figure 8.9 Newly Created Objects and Links

d. Analyzing the Impacts on the Design

Before the SHs are engaged in the formal deliberation process to choose among alternatives, the analyst with the designers analyze the impacts of resolving the issue using each alternative. For each alternative the analyst does the following:

1. Creates an object (PSDL component) for each proposed module. At this point this object has few attributes assigned real values. The most noticeable of these is a textual attribute describing in short the module functionality. Another (Boolean) attribute is *IsUnderAnalysis* which is assigned the value *True* indicating that this object is not and may not be part of the configuration. See sections C.4 and C.6 of chapterVI for the representation of these newly proposed components and their attributes.
2. Establishes the interdependency (if any) between modules affected by each alternative for both new and existing modules. These dependencies include *PartOf* and *UsedBy* relationships. If not established manually, the latter is computed by the system from the former.
3. Places the newly proposed modules in their proper location in the design hierarchy using *PartOf.*
4. Links each requirement component in the alternative to the module or modules it maps to. The link type is *UsedBy.*

The result of this process is shown in Figure 8.10 for the first alternative. The impacts of choosing the second alternative can be constructed in a similar way. The figure illustrates the mapping between the newly identified requirement components in the first alternative and the existing and/or proposed PSDL modules. As can be seen from the figure, taking this alternative affects not only the TDB functionality, but also the user interface. As an example, R1.5 maps to the root module (P4) of the user interface as well as to monitor_tracks module (P3.5) within TDBM. R1.5.1 maps to the modules P4.1 (an existing module) and P4.6 (proposed module). The rest of the mapping is shown in the figure which is expressed using *UsedBy* links.

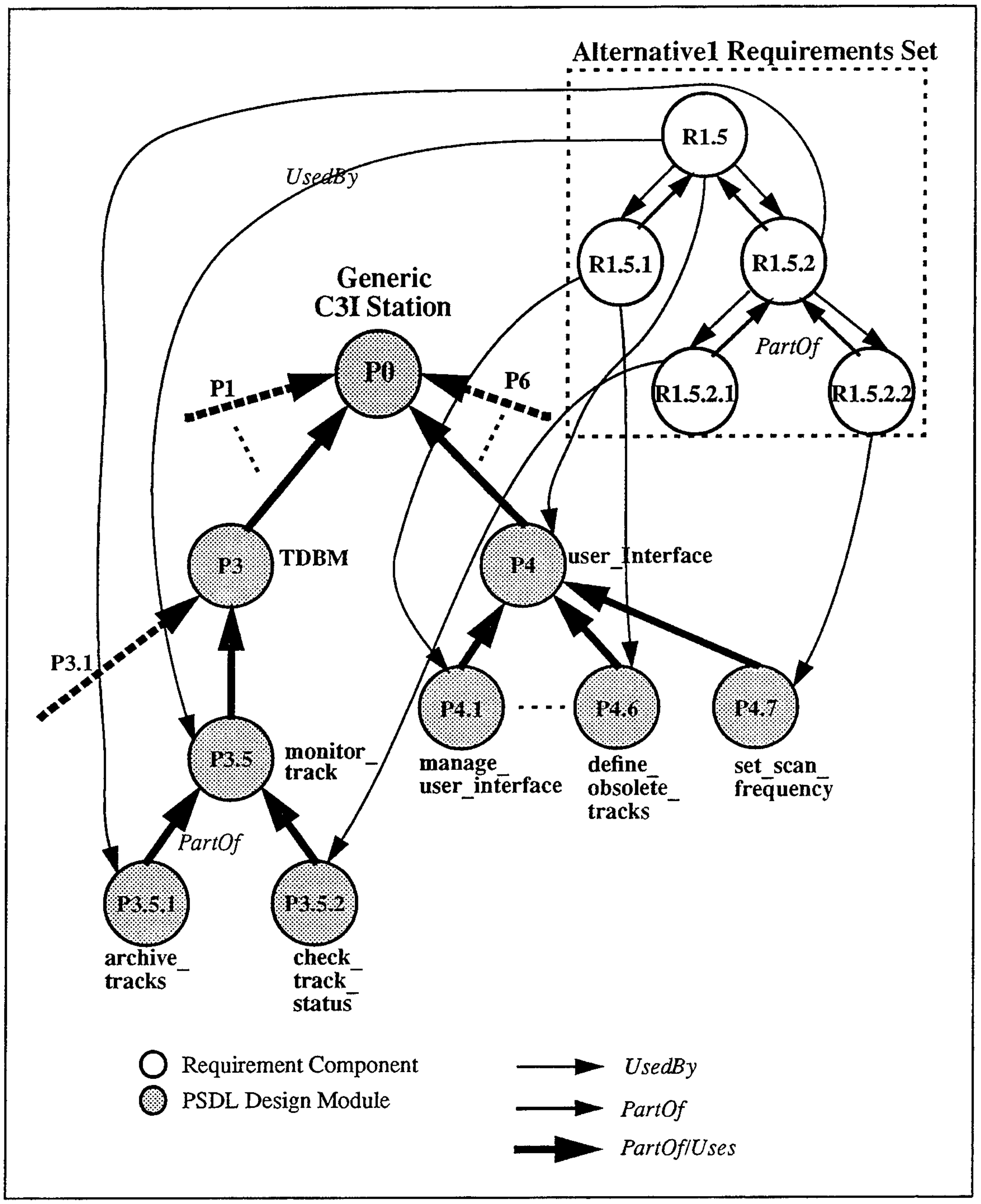


Figure 8.10 Requirements-Design Dependencies Related to Alternative 1.

e. Inference of New Relationships

Using the manually established relationships discussed above, the decision support mechanism infers more relationships. Rules introduced in Chapter IV are used here. From the manual links provided in Figure 8.10, and using the *PartOf* rule introduced in Chapter IV , (see page 65.), the mechanism infers the UsedBy relationships shown in Table 8.21. The PartOf rule has two variants: one applies to the requirements hierarchy and the other applies to the PSDL hierarchy. Both work in opposite directions. In requirements hierarchy *PartOf* and *UsedBy* relations have opposite direction. In PSDL hierarchy both have the same direction.

Component	Used by
R1.5	R1.5.1, R1.5.2
R1.5.2	R1.5.1.1, R1.5.1.2
P3	P3.5
P3.5	P3.5.1, P3.5.2
P4	P4.6, P4.7

Table 8.21. Inferred UsedBy

f. Supporting Information

At this point the available alternatives are determined along with the requirement components subset in each alternative. This subset is also linked to the affected PSDL modules; either existing as part of the current design or proposed to implement the proposed changes in requirements. The analyst supported by the system gathers the following information given in Table 8.22. This information is used to guide the SHs in the judgement process to be discussed in the following section.

The first alternative has five new requirement components. A total of eight PSDL modules are affected. Of these 3 are existing and require changes and 5 are newly proposed and require design and implementation. P3 needs to be changed to reflect the addition of a new child (P3.5). P4 needs to be changed to reflect the addition of two new children (P4.6 and P4.7). P4.1 needs to be changed to add a new user interface requirement. P3.5, P3.5.1, P3.5.2, P4.6, and P4.7 do not exist in the current design; they require design and implementation. The creation of P3.5 is within the context of the activity that decomposes P3. The creation of P3.5.1 and P3.5.2 is within the activity that decomposes P3.5. The same applies to the creation of P4.6 and P4.7 with respect to P4.

Analyzing the effort required to carry out the design and implementation, the analyst concludes that the first alternative needs 3 designers and the second needs four. This analysis is based on rough estimates of the required effort. In the first alternative one designer is needed to create the design of P3.5, decompose it into P3.5.1 and P3.5.2, and propagate the effect into P3. Another designer is needed to carry out the changes in P4.1 and propagate the effect into P4. A third designer is required to re-decompose P4 to add P4.6 and P4.7 and implement both. This adds up to a total of three designers. In a similar way of analysis the second alternative requires 4 designers to carry out the proposed design effort associated with this alternative.

As part of the analysis here is to determine the availability, field of expertise, and the expertise levels of the required designers. For alternative1 the analyst concluded that two major expertise areas are needed: database and user interface fields. One designer is required in the former and two in the latter. With this analysis information the analyst accesses the designers pool in the design database browsing for available designers with the required qualification. He found only two available of the required three. The analyst applied the same kind of analysis for alternative2 which led to the results shown in Table 8.22. Four designers are required, but only two are available with the required qualification.

Information	Alternatives	
	Alternative1	Alternativ2
No. of affected requirement components	5	3
No. of New PSDL modules	5	2
No. of Modified PSDL modules	3	4
No. of designers required	3	4
No. of designers available	2	2

Table 8.22. Supporting Information

E. FORMAL DEBATE SUPPORT

The support provided here concerns the SHs debate and the group decision making. SHs conduct their debate to choose one or more of the available alternative. They use our version of the IBIS model, Q-IBIS, quantified by the improved AHP. The outcome of the process is a decision that reflects the combined view points of all SHs. The final decision is transformed into a change request specifying what requirements to add or change and to what.

1. Inputs

The mechanism requires a ranking list of the participating SHs and a set of criteria. The ranking of the list is done as we explained in Chapter V. The set of criteria are used by the SHs to judge the available alternatives against. The SHs are also provided by the information gathered in the above section.

The set of criteria of judgement agreed upon are:

1. Budget: the relative impact of taking an alternative on the increase in the budget allocated.
2. Safety: the relative impact of taking an alternative on safety.
3. Deadline: the relative impact of taking an alternative on the delivery deadline of the system.

The ranking list for the five SHs attending the demonstration is given in Table 8.23. Ranks are given in decreasing order of importance.

Stakeholder	Rank
SH1	5
SH2	4
SH3	1
SH4	3
SH5	2

Table 8.23. The ranking List for SHs

2. Problem Structure

The decision problem of choosing among available alternatives by the five SHs is structured hierarchically into three levels as shown in Figure 8.11. The lowest (level 1) abstracts the decision question which is the focus of the debate. In our case it is choosing among the available alternatives. The second level down (level 2) includes criteria of judgement, in our case they are 3 as we discussed above. The third level includes the available alternatives for resolving the subject issue. Two alternatives are identified *Alt1* and *Alt2*. Since 5 SHs are attending, the system creates 5 instances of this structure. Each instance is presented to one of the SHs in a user friendly format to express his independent judgement on the available alternatives.

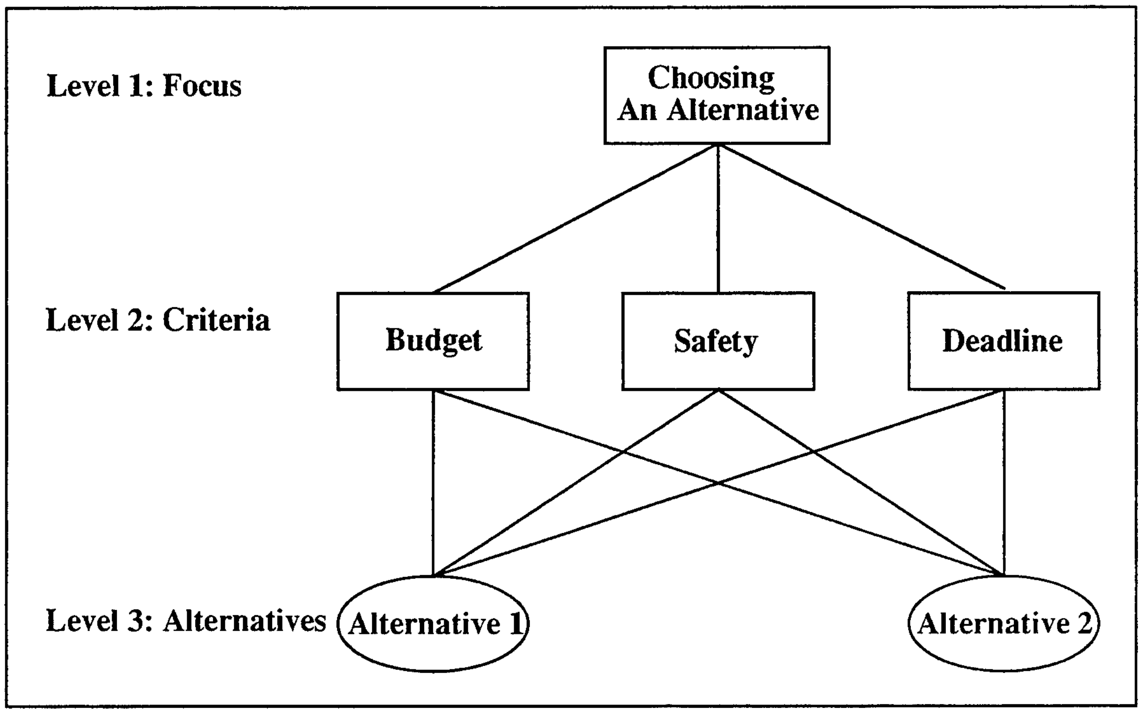


Figure 8.11 Hierarchy for Analyzing Alternatives in Phase2

3. Individual Judgements

The system presents each SH by the issue to be resolved, the available alternatives, and criteria of judgements, all annotated by related information that assists the SH. This information includes for example the relevant information gathered for each alternative. The SH reviews this information and can start his own judgement process. To keep the discussion focused, we only present here the judgement process related to the fourth SH in the list. The process is the same for the rest of them. The difference is in the outcome of the process. The complete results related to all SHs are included as appendix D. The rest of the individual judgement process for SH4 is given by the following steps:

1. SH4 compares each pair of the criteria and assigns a relative weight from the AHP scale to fill an importance matrix. This importance matrix corresponding to SH4 is given in Figure 8.12.

2. The system performs the AHP calculation on the matrix in 1. whose outcome is a priority vector. This vector provides a global ranking of all the criteria with respect to the *focus* (selecting an alternative(s)). The corresponding priority vector is shown in the same figure (Figure 8.12). The intermediate computations are not shown in the figure.

Choose Alternative	B	S	D	Priority Vector
B	1	1/3	2	0.27
S	2	1	4	0.62
D	1/3	1/3	1	0.15

Figure 8.12 Pairwise Comparison of Criteria and Priority Vector for SH4

3. SH4 compares the alternatives pairwise with respect to each one of the criteria. This comparison assigns a relative weight of each alternative over the other with respect to the criteria under consideration. Since we have three criteria, three importance matrices are filled in this step, one for each criteria. The three matrices that express the SH preference are given as part of appendix D.

4. The system performs the AHP calculation on each matrix in 3. to compute a priority vector of the alternatives with respect to each criterion. Each vector provides a global ranking of all the alternatives with respect to the corresponding criterion. The priority vector computed by the system for each matrix in 3. is included in appendix D.

5. The composite priority vector of the alternatives with respect to all criteria is computed as follows and is shown in Table 8.26. This vector represents the final judge-

ment of one SH (SH4).

- Form the priority matrix *L*, where *L* is an *nxm* matrix, *n* is the number of alternatives and *m* is the number of criteria. Each column is indexed by one of the criteria and each row is indexed by one of the alternatives (see Table 8.26). The matrix entries are formed from the priority vectors in 4. Each criterion's vector fills a column in the matrix.
- Multiply this matrix on the right by the vector obtained in 2.
- The result is the required composite priority vector that gives the ranking of all the alternatives with respect to all the criteria.
- The composite vector is the one used by the decision maker to select one or more of the alternatives.

These computation steps can be effectively performed by multiplying the priority of the alternatives under each criterion by the priority of the criterion (the bold face type entries in the top row of the table) and adding across criteria as shown in Table 8.26.

	Budget	Safety	Deadline	Composite Priorities
	(0.27)	**(0.62)**	**(0.15)**	
A1	0.37	0.50	0.40	0.47
A2	0.66	0.50	0.60	0.57

Table 8.24. Composite Priority Computation

The composite priorities give SH4 preference of the available alternatives. In our case the SH prefers the second alternative over the first one when both are evaluated against the given set of criteria. The preference is not very strong though: 47% for the first alternative and 57% for the second. When SH4 is done and these automatic computation outputs the results, the system creates an object of the QI_position and fills its attributes automatically by the vector of the composite priorities $(0.47, 0.57)^T$ and the four comparison matrices related to the SH. The system links this object automatically to the object of the SH and to any argument object this SH may have that justify his judgement. The QI_position object is also linked to the object of the issue under consideration, alternatives, and criteria.

This process is repeated for each of the five participating SHs. The complete comparisons matrices as well as the corresponding priority vectors and the composite priorities for the five SHs are included as appendix D. The next step is to combine these individual judgements into group one. This is discussed in the following section.

4. Group judgement

In support of the group final judgement, the system computes a group judgement. The system uses the already available composite priority vectors along with the priority vector computed from the ranking list of the SHs. For details concerning how these values are computed, refer to section D.3 of Chapter V. The results of this computation in our case are shown in Table 8.25 and Table 8.26. The priorities of the available alternatives as seen by the combined view points of all the SHs are given by the last column in Table 8.26. Each SH judgement is expressed by a column in this table. Each such column includes the importance of the SH (obtained from the ranking list), and the priority vector of the relative importance of the alternatives as judged by that SH. As can be seen from this table, the group prefers the second alternative over the first one. Perhaps in part due to the extra hardware required, the increase in the budget, the more housekeeping needed, the increase in the size and weight of the resulting system, and the expected delay in accessing obsolete information that may be needed as current one.

A QG_position object that carries this information is created and linked to the first issue. Any reasons behind this decision or other textual information goes into the QG_argument to be created for the whole group.

Stakeholder	Weight
SH1	0.33
SH2	0.27
SH3	0.07
SH4	0.20
SH5	0.13

Table 8.25. The priority vector of the SHs

	SH1	SH2	SH3	SH4	SH5	Group Priority
	(0.33)	**(0.27)**	**(0.07)**	**(0.20)**	**(0.13)**	
Alternative 1	0.47	0.45	0.52	0.36	0.48	0.46
Alternative 2	0.53	0.55	0.48	0.64	0.42	0.54

Table 8.26. The Combined Priority Matrix and the Group Priority Vector

5. Evolving Requirements and Design

According to the group final selection and their remarks and reasons, the analyst uses the system to create a change request (CR) object that represents the resolution of the issue. The CR object lists all impacted requirement components, either existing or newly proposed ones. For each such components both the old (if applicable) and the new statements of the component are cited. Since the final group decision has selected the second alternative, the CR contains requirement changes related to this alternative. In our case these changes are adding 3 new requirement components. The statements of these components are the same as was cited in section D. 1.b of this chapter. In other cases the CR may further refine the statements of the affected requirements. If this is the case, the analyst goes back to theses components, reflects any refinements, and modify the dependencies if necessary.

By finishing this task, the analyst advances the status of the analysis step associated with issue1 to *completed.* The step output is a modified or newly added set of requirement components. Accordingly the system automatically generates a sequence of proposed steps to propagate the changes in the requirements down into the affected design parts. The system uses the dependencies that tie the affected requirements and the design modules as well as the inter-dependencies within both the affected requirements and design modules. Figure 8.13 shows such dependencies for the selected alternative as kept in the design database and hence accessible by the system. Similar information related to the abandoned alternative is also kept in the design database as part of the design history.

The generated steps represent a proposed plan to carry out the work required by the CR. The manager reviews this plan, makes any necessary adjustments, specifies any management constraints and approves the plan to change the design. Table 8.27 below shows the automatically generated steps according to dependencies given in Figure 8.13.

There is also another set of induced steps in the design hierarchy to propagate changes from the lower to the next higher level. This set is not shown in the above table. These steps are primarily concerned with modifying the parent's specification as a result of modifying one or more of its children. Also they are concerned with modifying the implementation of the module whose specification is modified. For details regarding different types of these induced steps in the PSDL design hierarchy and management intervention for adjusting the generated plan, refer to [69].

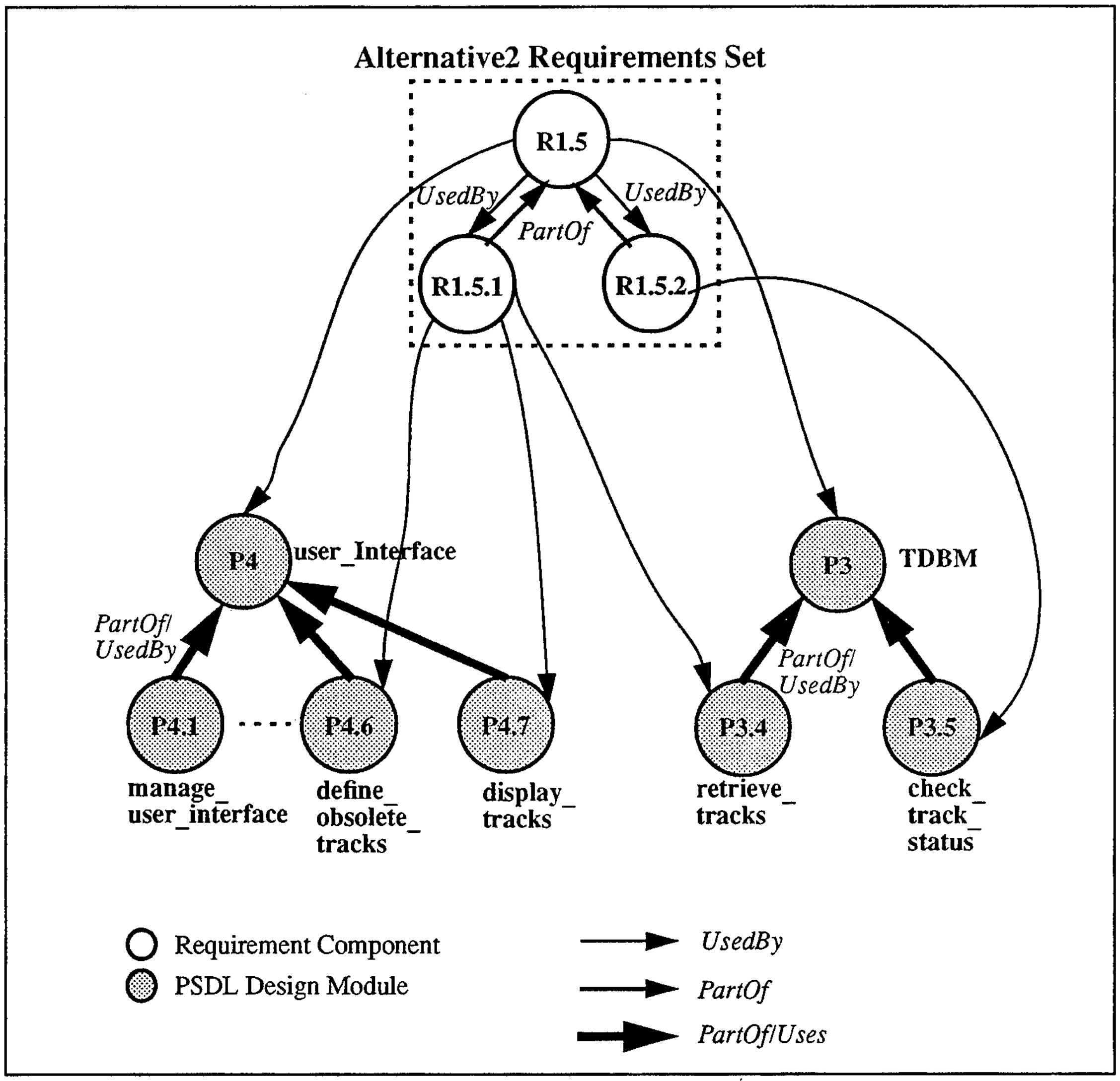


Figure 8.13 Requirements-Design Dependencies Related to Alternative 2.

Step Id	Secondary Input	Primary Input
s1	R1.5	P4
s2	R1.5.1	P4.6
s3	R1.5.1	P4.7
s4	R1.5	P3
s5	R1.5.1	P3.4
s6	R1.5	P3.5

Table 8.27. Automatically Generated Plan

IX. CONCLUSION AND DIRECTION FOR FUTURE RESEARCH

A. CONCLUSION

In this dissertation we have presented a formal model and a decision support mechanism for requirements analysis and evolution in the environment of computer-aided prototyping. The formal model and the decision support based on it provide the representation for requirements as they evolve, tied to the design specification and implementation. This enables any change in requirements to automatically expose the other affected parts of the system and automatically generate proposed plans to propagate the consequences of the requirements change into the system design and implementation. Propagating the consequences of requirement changes down into the design levels keeps changes consistent system wide and supports system evolution.

The model provides for representing the customers' responses to the demonstrated systems, synthesizing issues from these responses, analyzing the affected requirements, and assisting in identifying alternatives available to resolve open issues. This is done within the context of analysis and design activities generated automatically. These activities assist managers control and coordinate the project progress and implement projected plans.

We have also presented a formalism to be used in supporting stakeholders on their deliberation and judgement of the available alternatives for changing requirements. This latter support allows for independent judgement of each stakeholder and then combines these individual judgements into a decision that expresses the group point of view. The formalism provides a representation and supporting mechanism for capturing and recording design rationale. This can assist in design replay or justification of decisions as well as providing an important history trail for management references.

We also have developed a conceptual design for a project database. This database as reflected by the designed schema is capable of representing, storing and retrieving the design data and its rationale in a natural way. It also provides the capability of representing the complex and dynamically varying relationships that tie the system elements.

During the course of this research, we also developed a new implementation technique that allows Ada language to communicate smoothly with both C++ language and ONTOS DB functionality. This enables Ada programs such as the CAPS system to use a large part of the ONTOS DB functionality without system calls, process creation overhead, and file I/O for many complex data types. Within CAPS the new technique increases the productivity and uniformity of database applications development. This is caused by minimizing code portions written in C++ and relying on Ada in the rest. Without the new technique this could not be done. We made the development style of the minimized C++ and the maximized Ada code fixed and of standard format. In addition of increasing uniformity, this contributes to increased usability of the code fragments. This new technique has been tested by a database mock-up and was used successfully in developing the schema and application for the software base in CAPS.

B. SUGGESTIONS FOR FUTURE RESEARCH

One area that warrants further study is the area of quantifying stakeholder judgements of the available alternatives for resolving open issues. The quantification basis we provided is drawn from a quantification scale which is less realistic than ideally desired. What is needed is a more tangible measure of quantification.

Another important extension is to augment our process by a mechanism that checks consistency in requirements as new requirement components are added or existing ones are changed.

The above suggestion warrants pursuing again the limitation versus the added capability of using a deductive database model even with its limited representation for general knowledge. Using a deductive database model makes the tasks of inference and consistency checking readily available even with the fine granularity kind of knowledge which is more difficult to represent and check for consistency using e.g traditional database consistency constraints.

As a continuation of this work, the complete implementation of the model and the different supporting mechanisms is suggested. It is also worth the effort to complete the Ada binding for ONTOS DB to include all ONTOS functionality and facilities. The basis for these tasks have been laid by this dissertation which provides a complete template for both tasks.

LIST OF REFERENCES

[1] A. Czuchry, and D. Harries, "*KBRA: A NEW Paradigm for Requirements Engineering*," IEEE Expert, Winter 1988, pp. 21-35.

[2] A. Dardenne, A. Lamsweerde, S. Fickas, "*Goal-Directed Requirements Acquisition*," Science of Computer Programming, 20(1-2): 3-50, 1993.

[3] A. Sage, "*Decision Support Systems Engineering*," John Wiley & Sons, Inc., 1991.

[4] B. Greenspan, D. Maylopoulos, "*Knowledge Representation as Basis for Requirements Specification*," in Reading in AI and Software Engineering, (C. Rich and Richard C. Waters, Eds.), Morgan Kaufman, 1988.

[5] B. Ramesh, and Luqi, "*An Intelligent Assistant for Requirements Validation*," Journal of Systems Integration, Vol. 5, No. 2, 1995, pp. 157-177.

[6] B. Ramesh, and Luqi, "*Process Knowledge-Based Rapid Prototyping for Requirements Engineering*," Proceeding of the IEEE/ACM Symposium on Requirements Engineering, San Diego, CA, Jan 1993, pp. 248-225.

[7] B. Ramesh, and V. Dhar "*Representing and Maintaining Process Knowledge for Large-Scale Systems Development*," Proc. of the 6th annual Knowledge-Based Software Engineering Conf., Syracuse, NY, Sep 22-25, 1991, pp. 223-231.

[8] B. Ramesh, and V. Dhar "*Representing and Maintaining Process Knowledge for Large-Scale Systems Development*," IEEE Expert, April 1994, pp. 54-59.

[9] B. Ramesh, and V. Dhar "*Supporting Systems Development by Capturing Deliberations DuringRequirements Engineering*," IEEE Trans. on Software Engineering, Vol. 18, No. 6, June. 1992 pp. 448-510.

[10] B. Rao, "*Object-Oriented Databases: Technology, Applications, and Products*," McGraw-Hill, Inc, 1994.

[11] B. Shneiderman, "*Designing the User Interfac*e" Addison-Wesley Publishing Company, 1992.

[12] C. Date, "*Database Systems*," Vol I, Fifth ed., Addison-Wesley, 1990.

[13] C. Delobel, and F. Velez, "*Introduction to the System*," in Building an Object-Oriented Database System, (F. Bancilhon et. al, Eds.), Morgan Kaufmann Publishers, San Mateo, California, 1992, pp. 327-342.

[14] Chin_Liang Chang, "*Symbolic Logic and Mechanical Theorem Proving*," Academic Press, 1973.

[15] C. Pot, K. Takahasi, and A. Anton, "*Inquiry-Based Requirements Analysis,*" IEEE Software, March 1994, pp. 21-32.

[16] C. Rich, C. Waters, "*The Programming Apprentice*," Addison-Wesley, 1990.

[17] C. Rich, Y. Feldman, "*Seven Layers of Knowledge Representation and Reasoning in Support of Software Development*," IEEE Trans. Software Eng., Vol. 18, No. 6, June 1992, pp. 451-468.

[18] D. Dampier, "*A Formal Method for Semantics-Based Change-Merging of Software Prototypes*," Ph.D. Dissertation, Naval Psotgraduate School, Monterey, CA, Dec 1993.

[19] Doan Nguyen, "*An Architectural Model for Software Components Search*," Ph.D. Dissertation, Naval Psotgraduate School, Monterey, CA, Nov. 1995.

[20] D. Lowe, "*Co-opertative Structuring of Information: The Representation of Reasoning and Debate*," Int. J. Man-Machine Studies, Vol. 23, Aug. 1985, pp. 97-111.

[21] D. White, "*The Knowledge-Based Software Assistant: A Program Summery*," 6th annual Knowledge-Based Software Engineering Conf., Syracuse, NY, Sep 22-25, 1991, pp. 2-6.

[22] D. Wood, M. Christel, and S. stevens, "*A Multimedia Approach to Requirements Capture and Modeling*," Proceeding of the ICRE, April 18-22, Colorado Springs, CO, pp. 53-56.

[23] E. Byrne, "*IEEE Standards 830: Recommended Practice for Software Requirements Specifications*," Proceeding of the ICRE, April 18-22, 1994, Colorado Springs, CO, pp. 85.

[24] Finkelstein et al, "*Inconsistency Handling in Multiperspective Specification*," IEEE Trans. on Software Engineering, Vol. 20, No. 8, Aug. 1994, pp.569-578.

[25] G. Arang, L. Bruneau, and A. Feroldi, "*A Tool Shell for Tracking Design Decisions*," IEEE Software, March 1991.

[26] G. Booch, "*Software Components with Ada*," Benjamin/Cummings, 1987.

[27] G. Golub et al, "*Matrix Computations*," The Johns Hopkins University Press, Baltimore, MA, 1983.

[28] G. Moerkotte, and P. Lockemann, "*Reactive Consistency Control in Deductive Databases*," ACM Trans. Database Systems, Vol. 16, No. 4, Dec. 1991, pp. 670-702.

[29] G. Strange, "*Linear Algebra and its Applications*," Harcourt Brace Jovanovich, Inc, 3rd edition, 1988.

[30] H. Gallaire, J. Minker, and J. Nicolas, "*Logic and Database: A Deductive Approach,*" ACM Computing Surveys 16, No. 2, June 1984, pp. 153-185.

[31] H. Reubenstein and R. Waters, *"The Requirements Apprentice: Automated Assistance for Requirements Acquisition,"* IEEE Trans. Software Eng., Vol. 17, No. 3, March 1991, pp. 226-257.

[32] I. Mostov, *"A Model of Software Maintenance for Large Scale Military Systems"*, Master's Thesis, Naval Postgraduate School, Monterey, California, June. 1990.

[33] I Mostov, Luqi, and K. Hefner, *"A Graph Model of Software Maintenance"* Technical Report, NPS52-90-014, Dept. of Computer Science, Naval Postgraduate School, Monterey, California, Aug. 1989.

[34] J. Banerjee, et al., *"Data Model Issues for Object-Oriented Applications,"* ACM Transaction on Office Information Systems, 5(1), 1987.

[35] J. Conklin and M. Begeman, *"gIBIS: a hypertext tool for exploratory policy discussion,"* ACM Trans. Office Inform. Syst., Vol. 6, Oct. 1988, pp. 303-331.

[36] J. Doyle, *"A Truth Maintenance System,"* Artificial Intelligence, 1997, pp. 231-272.

[37] J. McDowell, *"A Reusable Component Retrieval System For Prototyping,"* Master's thesis, Dept. of Computer Science, Naval Postgraduate School, Monterey, California, 1991.

[38] J. Karlsson, *"Software Requirements Prioritizing,"* Proceedings of the Second International conf. on Requirements Engineering, Colorado Springs, Colorado, April 15-18, 1996, pp. 110-116.

[39] J. Linnerooth, *"Negotiating Environment Issues: A Role for the Analyst?"* in Effective Decision Support Systems, (J. Hawgood and P. Humphreys, Eds.), The Technical Press, 1987, pp. 33-48.

[40] J. Martins, M. Reinfrank (Eds.), *"Truth Maintenance Systems"* Spring-Verlag, Berlin, Germany, 1991.

[41] J. Minker, *"On Indefinite Databases and the Closed World Assumption,"* in Lecture Notes in Computer Science, No. 138, Spring-Verlag, 1982, pp. 292-308.

[42] J. Mylopoulos et al., *"Telos: Representing Knowledge About Information Systems,"* ACM Trans. on Information Systems, Vol. 8, No. 4, Oct. 1990, pp. 325-362.

[43] J. Rumbaugh et al, *"Object-Oriented Modelling and Design,"* Prentice-Hall Englewood Cliffs, N.J., 1991.

[44] K. Takahasi, et al, *"Hypermedia Support for Collaboration in Requirements Analysis"* Proceedings of the Second International conf. on Requirements Engineering, Colorado Springs, Colorado, April 15-18, 1996, pp. 31-40.

[45] L. Henschen, and H. Park, "*Compiling the GCWA in Indefinite Deductive Databases*," in Foundation of Deductive Databases and Logic Programming, (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 1988, pp. 395-438.

[46] Luqi, "*Computer-Aided Prototyping for a Command-And-Control System using CAPS*," IEEE Software, Jan. 1992, pp. 56-67.

[47] Luqi, "*A Graph Model for Software Evolution*," IEEE Transaction on Software Engineering. Vol. 16. No. 8. Aug. 1990.

[48] Luqi, J. Goguen, and V. Berzins, "*Formal Support for Software Evolution*" Proceeding of the 1994 Monterey Workshop on Software Evolution, Naval Postgraduate School, Monterey, CA, Sep. 7-9, 1994, pp. 13-21.

[49] Luqi and M. Ketabchi, "*A Computer Aided Prototyping System*," IEEE Software March 1988, pp. 66-72.

[50] Luqi, R. Steigerwald, G. Hughes, and V. Berzins, "*CAPS as a Requirement Engineering Tool*," Proceeding of Requirements Engineering and Analysis Workshop, Software Engineering Institute, Carneige Mellon University, Pittsburgh, PA, March 12-14, 1991, pp. 1-8.

[51] Luqi, V. Berzins, and R. Yeh, "*A Prototyping Language for Real-Time Software*," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988, pp.1409-1423.

[52] Luqi, V. Berzins, "*Rapidly Prototyping Real-Time Systems*," IEEE software 1988, pp. 25-36.

[53] Luqi, "*Software evolution via rapid prototyping*," IEEE computer, vol. 22, pp. 13-25, May 1989.

[54] Luqi, and Winston Royce, "*Status Report: Computer-Aided Prototyping*," IEEE Software, Nov. 1991, pp. 77-81.

[55] M. Alford "*Attacking Requirements Complexity using a Separation of Concerns*," Proceeding of the ICRE, April 18-22, 1994, Colorado Springs, CO, pp. 2-5.

[56] M. Atkinson, et al, "*The Object-Oriented Database System Manifesto*," in Building an Object-Oriented Database System, (F. Bancilhon et al, Eds.), Morgan Kaufmann Publishers, San Mateo, California, 1992, pp. 3-20.

[57] M. Birdie, and M. Jarke, "*On Integrating Logic Programming and Database*," in Expert Database Systems, (L. Kerschberg, Ed.), The Binjamin/Cummings, Menlo Park, CA, 1986, pp. 191-207.

[58] M. Davis, "*Applied Decision Support*," Prentice Hall, 1988.

[59] M. S. Feather, "*Constructing Specification by Combining Parallel Elaborations*," IEEE Trans. on Software Engineering, Vol. 15, No. 2, Feb. 1989, pp. 198-208.

[60] M. Loomis, "*Object Databases: The Essentials*," Addison-Wesley, 1995.

[61] M. Lubars, "*Representing Design Dependencies in an Issue-Based Style*," IEEE Software, July 1991, pp. 81-89.

[62] P. Bose, "*A Model for Decision Maintenance in the Win Win Collaboration Framework*," Proc. of the 10th annual Knowledge-Based Software Engineering Conf., Boston, Massachusetts, Nov. 12-15, 1995, pp. 105-113.

[63] P. Hsia, A. Davis, and D. Kung "*Status Report: Requirements Engineering*," IEEE Software, Nov. 1993, pp. 75-79.

[64] P. Hsia et al "*Formal Approach to Scenario Analysis*," IEEE Software, March 1994, pp. 33-40.

[65] R. Reiter, "*Towards a Logical Reconstruction of Relational Database Theory*," in reading in Artificial Intelligence and Database, (J. Mylopoulos and M Brodie, Eds.), Morgan Kaufmann, San Mateo, CA, 1989, pp. 301-327.

[66] R. Reiter, "*Deductive Question-Answering on Relational Databases*," in Logic and Databases, (H. Gallaire, and J. Minker, Eds.), Plenum Press, New York, 1987, pp. 149-177.

[67] R. Wen Hong, "*User Interface and Database Design for Software Database of the Computer-Aided Prototyping System (CAPS)*," Master Thesis, Computer Science Dept., Naval Postgraduate School, Monterey, California, March, 1996.

[68] S. Anderson, "*Functional Specification for a Generic C3I Station*," Master thesis, Computer Science Dept., Naval Postgraduate School, Monterey, CA, 1990.

[69] S. Badr, "*A Model and Algorithms for A Software Evolution Control System*," Ph.D. Dissertation, Computer Science Dept., Naval Postgraduate School, Monterey, CA, 1993.

[70] S. Badr, and Luqi, "*A Version and Configuration Model for Software Evolution*," Proceeding of the Fifth International Conference on Software Engineering and Knowledge Engineering, June 16-18, 1993, San Francisco, CA, pp. 225-227.

[71] S. Badr, and V. Berzins, "*A Software Evolution Control Model*" Proceeding of the 1994 Monterey Workshop on Software Evolution, Naval Postgraduate School, Monterey, CA, Sep. 7-9, 1994, pp.160-171.

[72] S. Dolgoff, "*Automated interface for retrieving reusable software components*," Master's thesis, Naval Postgraduate School, Monterey, California, 1992.

[73] S. Ian "*Software Engineering*," Fourth edition, Addison-Wesley, 1992.

[74] S. Schach " *Software Engineering*" Second edition, Aksen Associates, Inc., 1993.

[75] S. Sigfried, "*Understanding Object-Oriented Software Engineering*," IEEE Computer Society Press, 1996.

[76] T. Rose et al., "*A Decision-Based Configuration Process*" Software Eng. Journal, Sep. 1991, pp. 332-346.

[77] T. Saaty, and L. Vargas "*Prediction, Projection, and Forecasting*," Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[78] T. Saaty, "*Decision Making for Leaders: The Analytic Hierarchy Process for Decisions in a Complex World,*" RWS publications, Pittsburgh, PA, 2nd edition, 1990.

[79] T. Saaty, and L. Vargas, "*The Logic of Priorities: Application in Business, Energy, Health, and Transportation,*" Kluwer-Nighoff Publishing, Hingham, Massachusetts, 1982.

[80] T. Saaty, "*The Analytic Hierarchy Process,*" McGraw-Hill, Inc., 1980.

[81] U. Hahn, M. Jarke, and T. Rose "*Team Support in a Knowledge-Based Information Systems Environment,*" IEEE Trans. Software Eng., Vol. 17, May 1991, pp. 467-482.

[82] V. Berzins, and Luqi "*Software Engineering with Abstraction,*" Addison-Wesley, 1991.

[83] V.Berzins, Luqi, and A.Yehudai "*Using Transformation in Specification-Based Prototyping,*" IEEE Trans. on Software Engineering, Vol. 19, No. 5, May 1993, pp. 436-452.

[84] V. Coskun, and C. Kesoglu, "A Software Prototype for a Command, Control, Communication, and Intelligence (C3I) Workstation," Master thesis, Computer Science Dept., Naval Postgraduate School, Monterey, CA, 1990.

[85] V. Rajkovic at el, "*Ranking Multiple Options with DECMAK,*" in Effective Decision Support Systems, (J. Hawgood and P. Humphreys, Eds.), The Technical Press, 1987, pp. 49-60.

[86] W. Rzepka, J. Sidoran, and D. White, "*Requirements Engineering Technologies at Rome Laboratory,*" Proceeding of the IEEE/ACM Symposium on Requirements Engineering, San Diego, CA, Jan 1993, pp. 15-18.

[87] W. Yin, Luqi, and M. Tanik, "*Rapid Prototyping for Software Evolution,*" Technical Report, NPS52-90-009, Dept. of Computer Science, Naval Postgraduate School, Monterey, California, Aug. 1989.

[88] Z. Manna, R. Waldinger "*The Deductive Foundation of Computer Programming,*" Addison-Wesley, 1993.

[89] "*ONTOS DB 3.1 Developer's Guide,*" ONTOS, Inc., 1995.

[90] "*ONTOS DB 3.1 Reference Manual, Volume 1: Class*," ONTOS, Inc., 1995.

[91] "*TAE Plus User Interface Developer's Guide*," Version 5.3, Century Computing, Inc., Sep. 1993.

[92] "The Student Edition of MATLAB User's Guide," The Math Works, Inc., 1995.

APPENDIX A. TEMPLATE CODE

A. C++ Code

```
 /* ==============================================================

-- Unit            :class Person (.h)
-- File:person.h
-- Date            : Documented Oct 5,1995.
-- Author          :  Osman Ibrahim
-- Systems         :  Sun C++ and ONTOS (2.1)
-- Description     :The header for the Person Class that implements
                    the designer ADT in C++
 ==============================================================*/

#include <Object.h>

class Person : public Object
{
  private:
    int        priv_level; // The designer expertise level
// 0 : low
// 1 : Medium
// 2 : high

    int        priv_status; // The availability status of a designer
// 0 : free
// 1 : busy

  public:

    // Constructors
    Person(char* name=(char*)0,int level= 0, int status=0);

    Person (APL*);     // (Ontos required Constructor)

    // Ontos required member function)
    Type *getDirectType();

    // Accessors
    int      GetPersonLevel() ;

    void     SetPersonLevel(int level);

    int      GetPersonStatus();

    void     SetPersonStatus(int status);

    // OSMAN Jul 28, 1995

};

/* ==============================================================
-- Unit            :  class Person implementation (.cxx)
-- File            :  personc.cxx
-- Date            :  Documented Oct 5,1995.
-- Author          :  Osman Ibrahim
-- Systems         :  Sun C++ and ONTOS (2.1)
-- Description     :  Provides the implementation (definition) for the Person
                      Class that implements the designer ADT in C++
============================================================== */

#include "person.h"
#include <Directory.h>

//-----------------------------------------------
// constructors
//-----------------------------------------------

/* ------------------------------------------------------------------------ */

Person::Person(APL *theAPL) : Object(theAPL)
{
}

/* ------------------------------------------------------------------------ */

Person::Person(char* name,int level, int status): Object(name)

{
  initDirectType((Type *)OC_lookup("Person"));
  priv_level = level;
  priv_status= status;
}

//-----------------------------------------------
// accessors
//-----------------------------------------------

Type *Person::getDirectType()
{
  return (Type*)OC_lookup("Person");
}


/* ------------------------------------------------------------------------ */

void  Person::SetPersonLevel(int level)
{
  priv_level = level;
}

/* ------------------------------------------------------------------------ */

int Person::GetPersonLevel()
{
  return  priv_level;
}

/* ------------------------------------------------------------------------- */

void Person::SetPersonStatus(int status)
{
   priv_status = status;
}

/* ------------------------------------------------------------------------- */

int Person::GetPersonStatus()
{
  return priv_status;
}

/* ------------------------------------------------------------------------- */



/* ========================================================================

-- Unit              :  Header for db_utility (.h)
-- File              :  db_utility.h
-- Date              :  Documented Oct 5,1995.
-- Author            :  Osman Ibrahim
-- Systems           :  Sun C++ and ONTOS (2.1)
-- Description       :  Provide Function prototypes of some relevant Ontos DB
                        operation so that they are visible to and callable from
                        Ada. These operations needs to be extended in the future
                        in the same way as the need arises.

========================================================================= */

#include <Database.h>
#include "person.h"
#include "exception_interface.h"

int     ada_c_open_database(char* dbname);

void    ada_c_close_database(char* dbname);

void    ada_c_transaction_start();

void    ada_c_transaction_commit();

void    ada_c_save_to_db(Person*);

void    ada_c_delete_from_db(Person*);


/* --------------------------------------------------------------------------

-- Unit              :  The implementation for db_utility (.cxx)
-- File            :  db_utility.cxx
-- Date            :  Documented Oct 5,1995.
-- Author          :  Osman Ibrahim
-- Systems         :  Sun C++ and ONTOS (2.1)
-- Description     :  Provide the implementation for some relevant Ontos DB
                      operation so that they are visible to and callable from
                      Ada. These operations needs to be extended in the future
                      in the same way as the need arises.

---------------------------------------------------------------------------- */

#include "person_interface.h"
#include <Directory.h>
#include <Exception.h>
#include "exception_interface.h"

// Note : We tried to place all EXCEPTION objects in one header file (exception
// _interface.h) and be used whereever needed, but this resulted in an error
// came from the loader saying they are multiply defined, the same eror came
// out when even we placed them global in the same file, so we had to put each
// in the proper function where the Exception is expected.

// ===========================================================================

// An interface to ontos DB operation OC_open(dbname)

void   ada_c_open_database(char* dbname)
{
        ExceptionHandler  db_open_failed ("DatabaseOpenFailed");

        if (db_open_failed.Occurs())
               ada_c_set_exception_code(db_open_failed_code);
        else
  {
   ada_c_set_exception_code( normal_code);

            if (!(OC_dbIsOpen())) OC_open(dbname);
  }
}

// ============================================================================

// An interface to ontos DB operation OC_close(dbname)

void   ada_c_close_database(char* dbname)
{
        ExceptionHandler db_not_open    ("DatabaseNotOpen");

        if (db_not_open.Occurs())
               ada_c_set_exception_code(db_not_open_code);
        else
  {
            ada_c_set_exception_code( normal_code);

            OC_close(dbname);
  }
}

// ============================================================================

// An interface to ontos DB operation OC_transactionStart()

void   ada_c_transaction_start()

// According to Ontos; No Exceptions are associated with this operation
// Howerver the one added below wil catch any exception raised inside ONTOS

{

       ExceptionHandler ontos_falure  ("Failure");

       if (ontos_falure.Occurs())
               ada_c_set_exception_code(ontos_failure_code);
       else
           {
              ada_c_set_exception_code( normal_code);

           OC_transactionStart();
           }

}

// ============================================================================

// An interface to ontos DB operation OC_transactionCommit()

void   ada_c_transaction_commit()
{
        ExceptionHandler no_active_transaction ("NoTransaction");

        if (no_active_transaction.Occurs())
               ada_c_set_exception_code(no_active_transaction_code);
        else

           {
            ada_c_set_exception_code( normal_code);

            OC_transactionCommit();
           }
}

// ============================================================================

// I Think the following 2 operations should be moved to "person_inteface"
// because both are specific to designer objects and we can not make them
// general to accept any object type

// ============================================================================

// An interface to ontos DB operation putObject() .. specific to an object
```

```
void   ada_c_save_to_db(Person* ada_ptr)

{
  ExceptionHandler  object_already_exists("NameInUse");
  if (object_already_exists.Occurs())
              ada_c_set_exception_code(object_already_exists_code);
  else
     {
       ada_c_set_exception_code( normal_code);

       ada_ptr->putObject();
     }
}

// ===========================================================================

// An interface to ontos DB operation deleteObject() .. specific to an object


void   ada_c_delete_from_db(Person* ada_ptr)

{
  ExceptionHandler  no_such_object ("NameNotFound");

  if (no_such_object.Occurs())
              ada_c_set_exception_code(no_such_object_code);
  else
     {
       ada_c_set_exception_code( normal_code);

       ada_ptr->deleteObject();
     }
}

// ===========================================================================



/*=======================================================================

-- Unit           :  The header for exception_interface (.h)
-- File:  exception_interface.h
-- Date           :  Documented Oct 5,1995.
-- Author         :  Osman Ibrahim
-- Systems        :  Sun C++ and ONTOS (2.1)
-- Description    :  Provides an interface to a set of exception codes defined
                     below and the protypes for 2 functions that sets and gets
                     the values of an exception code set by differnt functions
                     from db_utility and person_interface units indicating
                     that some exception has occured or not, the meaning of
                     the exception codes are :

                     normal_code: no exception has occurred
                     object_already_exists_code:  An attempt was made to store
```

```
                        object into the DB but another object is
                        already exists in DB having the same name
                no_such_object_code: No such object in the DB
                        havining that name
                db_open_failed_code: An attempt was made to open
                        a DB that does not exist or using a wrong
                        name (eg misspelled)
                db_not_open_code: An attempt was made to close
                        a DB that was not previously opened
                no_active_transaction_code:   An attempt was made to
                        commit a transaction that was not started yet

============================================================ */

#include <Exception.h>

#define   normal_code                  0
#define   object_already_exists_code   1
#define   no_such_object_code          2
#define   db_open_failed_code     3
#define db_not_open_code  4
#define no_active_transaction_code 5
#define   ontos_failure_code           6

// Note: The following exception objects have been moved from here to the
//       proper local places; specifically each to a Function(s) inside
//       person_interface.cxx and db_utility.cxx. The reason for this
//       obligatory movement is because the loader complains of being defined
//       here and used there and gives me " objects so and so are multuply
//       defined". I'm not sure what is wrong because inspite of the error msg.,
//       the program links and run normally. OSMAN Oct 5, 1995

// ExceptionHandler  db_open_failed ("DatabaseOpenFailed");
// ExceptionHandler  db_open_failed("DatabaseOpenFailed");
// ExceptionHandler  db_not_open("DatabaseNotOpen");
// ExceptionHandler  no_active_transaction ("NoTransaction");
// ExceptionHandler  object_already_exists ("NameInUse");
// ExceptionHandler  no_such_object ("NameNotFound");


// The following function was introduced to allow ada to capture an exception
// raised inside ONTOS so that Ada can handle it in a way taht will not cause
// the program to abort because of a user error; e.g a misspelled DB name.

int ada_c_get_exception_code();

// The following function is used by different operations from inside
// db_utility and person_interface to set exception code into one of the
// above codes acording to the situation.

void ada_c_set_exception_code(int);


/*============================================================
-- Unit            :  The implementation for exception_interface (.cxx)
-- File            :  exception_interface.cxx
-- Date            :  Documented Oct 5,1995.
-- Author          :  Osman Ibrahim
-- Systems         :  Sun C++ and ONTOS (2.1)
-- Description     :  Provides the imlementation for the 2 functions set and get
                      exception_code that sets and gets the values of an
exception code set by differnt functions from db_utility
                      and person_interface units indicating that some exception
                      has occured or not, the meaning of the exception codes
                      are :

                      normal_code: no exception has occurred
                      object_already_exists_code:  An attempt was made to store
                                  object into the DB but another object is
                                  already exists having the same name
                      no_such_object_code: No such object in the DB
                                  havining that name
                      db_open_failed_code: An attempt was made to open
                                  a DB that does not exist or using a wrong
                                  name (eg misspelled)
                      db_not_open_code: An attempt was made to close
                                  a DB that was not previously opened
                      no_active_transaction_code:   An attempt was made to
                                  commit a transaction that was not started yet

============================================================= */

#include "exception_interface.h"

int global_exception_code = 0 ;

int ada_c_get_exception_code()

// The following function was introduced to allow ada to capture an exception
// raised inside ONTOS so that Ada can handle it in a way taht will not cause
// the program to abort because of a user error; e.g a misspelled DB name.

{
   return global_exception_code;
}

// The following function is used by different operations from inside
// db_utility and person_interface to set exception code into one of the
// above code acording to the situation.

void ada_c_set_exception_code(int exception_code)

{
global_exception_code = exception_code ;
}
```

```
/* ==========================================================================

-- Unit            :Header for person_interface (.h)
-- File            :  person_interface.h
-- Date            :  Documented Oct 5,1995.
-- Author          :  Osman Ibrahim
-- Systems         :  Sun C++ and ONTOS (2.1)
-- Description     :  The sole reason for this unit is to allow ADA to Create,
                      Access, and manipulate objects (instances) of the
                      "Person Class". We tried to do that directly without that
                      second level interface, but we did not succeed.
                      Comments about how ada can communicate with code written
                      in C++ can be found in some of the Ada files in this Dir.
========================================================================== */

#include <Database.h>
#include <Directory.h>
#include "person.h"
#include <Type.h>


// OSMAN Jul 28, 1995

// The following operation is just for making ada able to call the constructors
// of the Person Class.

Person*  ada_c_new_designer( char* Myname , int Mylevel , int Mystatus );

// The following operation is to allow Ada to lookup and retrieve an instance
// of the Person Class

Person*  ada_c_find_designer(char*);

// The following Operations each coresponds to a member function of the Person
// Class.

char*    ada_c_GetPersonName(Person* ada_ptr);

void     ada_c_SetPersonName(char* name, Person* ada_ptr);

int     ada_c_GetPersonLevel(Person*) ;

void    ada_c_SetPersonLevel(int level, Person*);

int     ada_c_GetPersonStatus(Person*);

void    ada_c_SetPersonStatus(int status, Person*);


//---------------- DESIGNER ITERATOR--------------------

// The following operations lumps an iterator suitable for looping through
// instances of the Person class and returning each of these instances
// The following syntax of the Instance Iterator, although is given for
// the Person Type, it is general enough to apply to any other TYPE under
// condititlon the Type MUST be classified into the DB (using Ontos CLASSIFY
// utility) with the +X switch so that Ontos will maintain an aggregate of all
// instance of that TYPE, in this case the Type is called "has an EXTENSION"
//  .. refer to ONTOS DB Tools and Utilities Guide Ch3.
// WATCH OUT though that using +X swith has a perfomance degredation penality,
// it slows down the application.

InstanceIterator* ada_c_Create_Instance_Iterator(char* type_name);

Person*            ada_c_Get_Next_Element(InstanceIterator* it);

void               ada_c_Reset_Iterator(InstanceIterator* it, char* type_name);

void               ada_c_Destroy_Iterator(InstanceIterator* it);

OC_Boolean         ada_c_Has_More_Elements(InstanceIterator* it);



/* =================================================================

-- Unit             :  The implementation for person_interface (.cxx)
-- File             :  person_interface.cxx
-- Date             :  Documented Oct 5,1995.
-- Author           :  Osman Ibrahim
-- Systems          :  Sun C++ and ONTOS (2.1)
-- Description      :  The sole reason for this unit is to allow ADA to Create,
                       Access, and manipulate objects (instances) of the
                       "Person Class". We tried to do that directly without that
                       second level interface, but we did not succeed.
                       Comments about how ada can communicate with code written
                       in C++ can be found in some of the Ada files in this Dir.
============================================================== */

#include "person_interface.h"
#include <Exception.h>
#include "exception_interface.h"


// OSMAN Jul 28, 1995
// revised Sep 29, 1995 to incorprate Exceptions

// Note : We tried to place all EXCEPTION objects in one header file (exception
// _interface.h) and be used whereever needed, but this resulted in an error
// came from the loader saying they are multiply defined, the same error came
// out when even we placed them global in the same file, so we had to put each
// in the proper function where the Exception is expected.

//=====================================================================

// The following operation is just for making ada able to call the constructors


Person* ada_c_new_designer( char* Myname,int Mylevel,int Mystatus)
```

```
{

  ExceptionHandler  object_already_exists("NameInUse");

  if (object_already_exists.Occurs())
               ada_c_set_exception_code(object_already_exists_code);
  else
     {
  ada_c_set_exception_code( normal_code);

  Person* aperson =  new Person(Myname, Mylevel, Mystatus);

  return aperson;
     }
}

//===========================================================================

Person* ada_c_find_designer(char* person_name)

{

  ExceptionHandler  no_such_object ("NameNotFound");

  if (no_such_object.Occurs())
              ada_c_set_exception_code(no_such_object_code);
  else
     {
 ada_c_set_exception_code( normal_code);

 Person *aPerson = (Person*)OC_lookup(person_name);

    return aPerson;
     }

}

//===========================================================================

char*  ada_c_GetPersonName(Person* ada_ptr)

{
  ExceptionHandler  no_such_object ("NameNotFound");

  if (no_such_object.Occurs())
              ada_c_set_exception_code(no_such_object_code);
  else
     {
       ada_c_set_exception_code( normal_code);

 return ada_ptr->Name();
     }
}

//===========================================================================
void  ada_c_SetPersonName(char* name, Person* ada_ptr)

{
  ExceptionHandler  no_such_object ("NameNotFound");

  if (no_such_object.Occurs())
              ada_c_set_exception_code(no_such_object_code);
  else
     {
       ada_c_set_exception_code( normal_code);

 ada_ptr->Name(name);
     }
}

//==========================================================================

int     ada_c_GetPersonLevel(Person* ada_ptr)

{
  ExceptionHandler  no_such_object ("NameNotFound");

  if (no_such_object.Occurs())
              ada_c_set_exception_code(no_such_object_code);
     {
       ada_c_set_exception_code( normal_code);

       ada_c_set_exception_code( normal_code);

 return ada_ptr->GetPersonLevel();
     }
}

//==========================================================================

void  ada_c_SetPersonLevel(int level, Person* ada_ptr)

{
  ExceptionHandler  no_such_object ("NameNotFound");

  if (no_such_object.Occurs())
              ada_c_set_exception_code(no_such_object_code);
  else
     {
       ada_c_set_exception_code( normal_code);

 ada_ptr->SetPersonLevel(level);
     }
}

//==========================================================================

int     ada_c_GetPersonStatus(Person* ada_ptr)
```

```
{
  ExceptionHandler  no_such_object ("NameNotFound");

  if (no_such_object.Occurs())
              ada_c_set_exception_code(no_such_object_code);
  else
     {
       ada_c_set_exception_code( normal_code);

 return ada_ptr->GetPersonStatus();
  }
}

//==========================================================================

void    ada_c_SetPersonStatus(int status, Person* ada_ptr)

{
  ExceptionHandler  no_such_object ("NameNotFound");

  if (no_such_object.Occurs())

              ada_c_set_exception_code(no_such_object_code);
  else
     {
       ada_c_set_exception_code( normal_code);

 ada_ptr->SetPersonStatus(status);
  }
}

//==========================================================================

//------------------ DESIGNER ITERATOR--------------------

 InstanceIterator* ada_c_Create_Instance_Iterator(char* type_name)

{
   InstanceIterator* it = new  InstanceIterator((Type*) OC_lookup(type_name));
   return it;
}

Person* ada_c_Get_Next_Element(InstanceIterator* it)

{
  Person* next_person = (Person*) (Entity*)it->operator()();
  return next_person;
}

void ada_c_Reset_Iterator(InstanceIterator* it, char* type_name)

{
     it->Reset((Type*) OC_lookup(type_name));
}
```

```
void ada_c_Destroy_Iterator(InstanceIterator* it)

{
     it->Destroy();
}

OC_Boolean ada_c_Has_More_Elements(InstanceIterator* it)

{
  return it->moreData();
}

//
==============================================================
==
```

B. Ada Source Code

```
==============================================================
=======
-
--
-- Component:  designer_PKG Spec
-- Author:  Osman Ibrahim
-- Date:        SEP. 1995
-- Language:  Ada
-- Compiler:  caps-suns7 SunAda
-- Purpose:  This package spec represents an Ada mirrored image for the C++
--             class "Person". It encpsolates all operations and types defined
--             over designer in a way making an abstract DT of it.It also
--             defines the interface between Ada operations defined over
--             designer and the coresponding C++ operations.
--             Procedure names are given followed by their interface
--             (C in all cases) and interface names. The a.ld pre-link
--             link_with pragmas are given in the file link_with_pragmas.a

--             Refer to this PKG body for some other detail regarding the
--             Conventions used for subprograms names

--             The method I used to interface Ada to a C++ code is as follows:

--             1- compile your C++ code you like to interface Ada to using a
--                C++ compiler

--             2- use the Unix nm command to get the symbolic name of the
--                subroutine you like to link your Ada to using
--                "pragma_interface" and pragme_interface_name. Choose the
--                symbolic name which is preceeded by "T". Use the symbolic
--                name in the pragma interface_name and the C++ subroutine
--                  name in the pragma interface.

--               3- use link_with_pragma to pre-link with the C++ object code
--                  in each of the files containing that code (.o) files. These
--                  link with pragmas are included in the file
--                  link_with_pragmas.a for this experimental application
--                  template

-- Example:  Suppose you have an Ada function called ada_new_designer and
--              you have a C++ function that implements the Ada function called
--              c_new_designer and the object code for the C++ function is
--              in the file person_inteface.o and now you want to interface the
--              Ada function to the coresponding C++ function :
--
--           1- Assuming Your C++ code is already compiled and you have the (.o)
--              file(s).

--           2 use nm:
--                     >nm -ao person_inteface.o | grep "c_new_designer"
--               - The output will be  a punch of names as follows:

-- person_interface.o:000006c8 T _c_new_designer__FPciT2
-- person_interface.o:00000000 - 00 0000  LSYM c_new_designer__FPciT2 :ZtF
-- person_interface.o:000006c8 - 00 001c   FUN c_new_designer__FPciT2:F(0,1

--              - Choose the first one (preceeded by "T"):
--                       "_c_new_designer__FPciT2" as the symbolic name.
--
--             Note : for details about why this symbolic name look this
--                    strange,  refer to a paper by Bjarne Stroustrup
--                    titled "Type-safe Linkage for C++"... by the way C++
--                    is one of his contributions.

--               - Now in your ada code, all you need is the following

--                  Function ada_new_designer(name      : in c_string;
--                                       level     : in integer ;
--                                       status    : in integer   )
--                                                     return designer;

--              pragma interface(C,c_new_designer);  -- C++ subroutine name
--              pragma interface_name(c_new_designer, "_c_new_designer__FPciT2");

--                 - This way you got an Ada function called ada_new_designer
--                   so when you say somewhere in you Ada code:
--                   ada_new_designer(some_name,some_integer,another_integer)
--                   you are actually calling the coresponding C++ function:
--                   c_new_designer

--          3- Now for step #3 you have 3 ways to link with the C++ relevant
--             object files :
--                        - use withn  (not recommended by Ada documentations)
--                        - supply the C++ (.o) files  in the command
--                          line as opetions to a.ld , this is doable and
--                          can be included in the Makefile
--                          - The way I'm using is to use link_with pragma to
--                            link to the desired (.o) file

--                             ex: pragma link_with("person_interface.o");
--
--                 I've included all such pragma link_with in the file
--                 link_with_pragmas.a and then my top level driver (interface)
--                 includes a "with link_with_pragmas_PKG"


--
===============================================================================
======

with a_strings;
use  a_strings;

with exception_interface_PKG;
use  exception_interface_PKG;

package designer_PKG is

-- OSMAN 9/29

   type designer_record is private;

   type designer is access designer_record;

   type expertise_level is (low, medium, high); --Expertise level for a designer

   type status          is (free, busy);        -- Designer Availability status

   Function new_designer(name      : in a_string;
                         level     : in expertise_level;
                         s         : in status         ) return designer;

   Function get_designer(name      : in a_string) return designer;

   Function GetPersonName(d  : in designer) return a_string;

   procedure SetPersonName(Name    : in a_string;
                           d       : in designer  );

   Function GetPersonLevel(d       : in designer) return expertise_level;

   procedure SetPersonLevel(level : in expertise_level;
                            d     : in designer         );

   Function GetPersonStatus(d      : in designer) return status;

   procedure SetPersonStatus(s     : in status;
                             d     : in designer  );

procedure PutDesigner(d      : in designer );
```

```
---------------------------------------------------------------------

-- DESIGNER ITERATOR

-- The following operations lumps an iterator suitable for looping through
-- instances of the Person class and returning each of these instances
-- The following syntax of the Instance Iterator, although is given for
-- the Person Type, it is general enough to apply to any other TYPE under
-- condititon the Type MUST be classified into the DB (using Ontos CLASSIFY
-- utility) with the +X switch so that Ontos will maintain an aggregate of all
-- instance of that TYPE, in this case the Type is called "has an EXTENSION"
--  .. refer to ONTOS DB Tools and Utilities Guide Ch3.
-- WATCH OUT though that using +X swith has a perfomance degredation penality,
-- it slows down the application.

-- The syntax of using this iterator is as follows

--       Designer_Iter := CREATE_ITERATOR ( type_name );

--       While Has_More_Elements(Designer_Iter) loop

--       Designer_Object := Get_Next_Element(Designer_Iter)

--             do something with Designer_Object
--             .....
--             ....

--       end loop


--           You can also issue:

--             RESET_ITERATOR(Designer_Iter) : to re-iterate

--             DELETE_ITERATOR(Designer_Iter) : to deallocate memory

--   Note also that Designer_Iter and Designer_Object are of the same
--    type: designer of this PKG



   Function Create_Instance_Iterator(type_name : in a_string ) return designer;

   Function Get_Next_Element(d : in designer ) return designer;

   procedure Reset_Iterator(d : in designer; type_name : in a_string );

   procedure Destroy_Iterator(d : in designer );

   Function Has_More_Elements(d : in designer ) return BOOLEAN;

---------------------------------------------------------------------
```

```
-- EXCEPTIONS

-- The 2 Exceptions "no_such_object" and "object_already_exists" applies to
-- all persistent types, that is why they are included in a separate PKG
-- "exception_interface_PKG", so they can be visible to all types.
-- Renaming both here is a matter of readability to know that we are
-- talking about Designer Objects.

  no_such_designer         : exception renames no_such_object;

  designer_already_exists : exception renames object_already_exists;

------------------------------------------------------------------------

  private
        type designer_record is record
                null;
        end record;

end designer_PKG;


===============================================================
======
--
-- Component:  designer_PKG Body
-- Author:     Osman Ibrahim
-- Date:       SEP. 1995
-- Language:   Ada
-- Compiler:   caps-suns7 SunAda
-- Purpose:    This package Body represents an Ada mirrored image for the C++
--             class "Person". It encpsolates all operations and types defined
--             over designer in a way making an abstract DT of it.It also
--             defines the interface between Ada operations defined over
--             designer and the coresponding C++ operations.
--             Procedure names are given followed by their interface
--             (C in all cases) and interface name. The a.ld pre-link
--             link_with pragmas are also given.

--             Naming Conventions used for subprograms names are as follows:

--                1. Subprograms interfacing to coresponding C++ subprograms
--                   have the same name as the the coresponding C++ subprograms
--                   and these names are identified by the prefix "ada_c_"

--                2. The Ada subprograms implementing the functionality of the
--                   subprograms in 1. and do not include the interface detail
--                   like pragmas and c_strings are given the same name as the
--                   coresponding subprograms in 1. above without the prfix.

--             Refer to this PKG Spec for some other detail regarding the way
--             I used to interface Ada to a C++ code.
--
===============================================================
======

with a_strings;
use  a_strings;

with c_strings;
use  c_strings;

with db_utility_PKG;
use  db_utility_PKG;

with text_io;
use  text_io;

with exception_interface_PKG;
use  exception_interface_PKG;

package body designer_PKG is

-- OSMAN 9/29

-------------------------------------------------------------------------------


   Function ada_c_new_designer(name       : in c_string;
                               level      : in integer ;
                               status     : in integer) return designer;

     pragma interface(C,ada_c_new_designer);
     pragma interface_name(ada_c_new_designer, "_ada_c_new_designer__FPciT2");

-------------------------------------------------------------------------------


   Function new_designer(name       : in a_string;
                         level      : in expertise_level;
                         s          : in status           ) return designer is

   C_name       : c_string := to_c(name);
   d            : designer;

   begin

       d := ada_c_new_designer(C_name, expertise_level'POS(level),
                               status'POS(s)                       );
       case get_exception_code is
when normal_code    =>
               null;
               when object_already_exists_code      =>
                      raise designer_already_exists;
               when others                          =>
                      raise ONTOS_Failure;
       end case;

       return d;
```

```
end new_designer;

--------------------------------------------------------------------------------

Function ada_c_find_designer(name      : in c_string) return designer;

  pragma interface(C,ada_c_find_designer);
  pragma interface_name(ada_c_find_designer, "_ada_c_find_designer__FPc");

--------------------------------------------------------------------------------

Function get_designer(name      : in a_string) return designer is

C_name       : c_string := to_c(name);
d            : designer;

begin

    d := ada_c_find_designer(C_name);

    case get_exception_code is
            when normal_code                   =>
                   null;
            when no_such_object_code           =>
                   raise no_such_designer;
            when others                        =>
                   raise ONTOS_Failure;
    end case;

    return d;

end get_designer;


--------------------------------------------------------------------------------
Function ada_c_GetPersonName(d  : in designer) return c_string;

  pragma interface(C,ada_c_GetPersonName);
  pragma interface_name(ada_c_GetPersonName,
                        "_ada_c_GetPersonName__FP6Person");

--------------------------------------------------------------------------------


Function GetPersonName(d  : in designer) return a_string is

a_name       : a_string := to_a(ada_c_GetPersonName(d));

begin

    case get_exception_code is
            when normal_code                   =>
                   null;
            when no_such_object_code           =>
```

```
                raise no_such_designer;
         when others                        =>
                raise ONTOS_Failure;
    end case;

    return a_name;

end GetPersonName;

--------------------------------------------------------------------------------

procedure ada_c_SetPersonName(
                 Name    : in c_string;
                 d       : in designer);

  pragma interface(C,ada_c_SetPersonName);
  pragma interface_name(ada_c_SetPersonName,
                        "_ada_c_SetPersonName__FPcP6Person");
--------------------------------------------------------------------------------

procedure SetPersonName(Name    : in a_string;
                        d       : in designer  ) is

C_name      : c_string := to_c(Name);

begin

    ada_c_SetPersonName(C_name, d);

    case get_exception_code is
         when normal_code                   =>
                null;
         when no_such_object_code           =>
                raise no_such_designer;
         when others                        =>
                raise ONTOS_Failure;
    end case;

end SetPersonName;

--------------------------------------------------------------------------------

Function ada_c_GetPersonLevel(d  : in designer) return integer;

  pragma interface(C,ada_c_GetPersonLevel);
  pragma interface_name(ada_c_GetPersonLevel,
                         "_ada_c_GetPersonLevel__FP6Person");

--------------------------------------------------------------------------------

Function GetPersonLevel(d  : in designer) return expertise_level is

level_code : integer;
```

```
   level      : expertise_level;

   begin

      level_code := ada_c_GetPersonLevel(d);

      case level_code is
               when expertise_level'POS(low)    =>
                    level := low;

               when expertise_level'POS(medium) =>
                    level := medium;

               when expertise_level'POS(high)   =>
                    level := high;

               when others                      =>
null;
               -- Is this last line OK ???
-- i.e will not cause any problems; i.e is it safe??
-- I thought of constraining level_code to be of
-- 0 .. 2 range; but this may cause problems in the
-- C++ side.. I'm not sure.
      end case;

      case get_exception_code is
               when normal_code                  =>
                      null;
               when no_such_object_code          =>
                      raise no_such_designer;
               when others                       =>
                      raise ONTOS_Failure;
      end case;


      return level;

   end GetPersonLevel;

-------------------------------------------------------------------------------

   procedure ada_c_SetPersonLevel(
                    level : in integer;
                    d     : in designer);

     pragma interface(C,ada_c_SetPersonLevel);
     pragma interface_name(ada_c_SetPersonLevel,
                           "_ada_c_SetPersonLevel__FiP6Person");


-------------------------------------------------------------------------------

   procedure SetPersonLevel(level : in expertise_level;
                           d     : in designer) is
   begin
       case level is
when low      =>
                         ada_c_SetPersonLevel(expertise_level'POS(low) , d);

              when medium    =>
                   ada_c_SetPersonLevel(expertise_level'POS(medium) , d);

              when high      =>
                   ada_c_SetPersonLevel(expertise_level'POS(high) , d);

      end case;

      case get_exception_code is
              when normal_code                 =>
                     null;
              when no_such_object_code         =>
                     raise no_such_designer;
              when others                      =>
                     raise ONTOS_Failure;
       end case;

   end SetPersonLevel;



-------------------------------------------------------------------------------

   Function ada_c_GetPersonStatus(d      : in designer) return integer;

     pragma interface(C,ada_c_GetPersonStatus);
     pragma interface_name(ada_c_GetPersonStatus,
                           "_ada_c_GetPersonStatus__FP6Person");

-------------------------------------------------------------------------------

   Function GetPersonStatus(d       : in designer) return status is

   status_code : integer;
   s           : status;

   begin

      status_code := ada_c_GetPersonStatus(d);

      case status_code is
              when status'POS(free)    =>
                   s := free;

              when status'POS(busy)    =>
                   s := busy;

              when others              =>
                    null;
              -- Is this last line OK ???
              -- i.e will not cause any problems; i.e is it safe??
              -- I thought of constraining level_code to be of
              -- 0 .. 2 range; but this may cause problems in the
              -- C++ side.. I'm not sure.


      end case;

      case get_exception_code is
              when normal_code                  =>
                      null;
              when no_such_object_code          =>
                      raise no_such_designer;
              when others                       =>
                      raise ONTOS_Failure;
      end case;

      return s;

   end GetPersonStatus;


--------------------------------------------------------------------------------

   procedure ada_c_SetPersonStatus(
                    Status : in integer;
                    d      : in designer);

     pragma interface(C,ada_c_SetPersonStatus);
     pragma interface_name(ada_c_SetPersonStatus,
                          "_ada_c_SetPersonStatus__FiP6Person");

--------------------------------------------------------------------------------

   procedure SetPersonStatus(s      : in status;
                             d      : in designer) is
   begin

       case s is
              when free     =>
                   ada_c_SetPersonStatus(status'POS(free), d);

              when busy     =>
                   ada_c_SetPersonStatus(status'POS(busy), d);
      end case;

       case get_exception_code is
              when normal_code                  =>
                      null;
              when no_such_object_code          =>
                      raise no_such_designer;
              when others                       =>
                      raise ONTOS_Failure;
       end case;
   end SetPersonStatus;

-------------------------------------------------------------------------------

   procedure PutDesigner(d      : in designer) is

   -- The following 2 instantiations are for outputing
-- expertise_level and status values respectively

   package level_enum_io is new ENUMERATION_IO(expertise_level);
   use level_enum_io;

   package status_enum_io is new ENUMERATION_IO(status);
   use status_enum_io;


begin

      transaction_start;

         PUT("Person Name is ");
         PUT(GetPersonName(d).s);
         NEW_LINE;

         PUT("Person Level is ");
         PUT(GetPersonLevel(d));
         NEW_LINE;

         PUT("Person Status is ");
         PUT( GetPersonStatus(d));
         NEW_LINE;

      transaction_commit;

   end PutDesigner;

-------------------------------------------------------------------------------

-- DESIGNER ITERATOR

   Function ada_c_Create_Instance_Iterator(type_name : in c_string)
                        return designer;

     pragma interface(C,ada_c_Create_Instance_Iterator);
     pragma interface_name(ada_c_Create_Instance_Iterator,
                           "_ada_c_Create_Instance_Iterator__FPc");

-------------------------------------------------------------------------------


   Function Create_Instance_Iterator(type_name : in a_string)
return designer is
begin
return ada_c_Create_Instance_Iterator(to_c(type_name));
```

```
end Create_Instance_Iterator;

------------------------------------------------------------------------------

   Function ada_c_Get_Next_Element(d : in designer) return designer;

     pragma interface(C,ada_c_Get_Next_Element);
     pragma interface_name(ada_c_Get_Next_Element,
                          "_ada_c_Get_Next_Element__FP16InstanceIterator");

------------------------------------------------------------------------------


   Function Get_Next_Element(d : in designer) return designer is

begin
return ada_c_Get_Next_Element(d);

end Get_Next_Element;

------------------------------------------------------------------------------


   procedure ada_c_Reset_Iterator(d : in designer; type_name : in c_string);

     pragma interface(C,ada_c_Reset_Iterator);
     pragma interface_name(ada_c_Reset_Iterator,
                          "_ada_c_Reset_Iterator__FP16InstanceIteratorPc");

------------------------------------------------------------------------------


   procedure Reset_Iterator(d : in designer; type_name : in a_string) is

begin
ada_c_Reset_Iterator(d, to_c(type_name));

end Reset_Iterator;


------------------------------------------------------------------------------


   procedure ada_c_Destroy_Iterator(d : in designer);

     pragma interface(C, ada_c_Destroy_Iterator);
     pragma interface_name( ada_c_Destroy_Iterator,
                           "_ada_c_Destroy_Iterator__FP16InstanceIterator");

------------------------------------------------------------------------------


   procedure Destroy_Iterator(d : in designer) is

begin
```

```
ada_c_Destroy_Iterator(d);

end Destroy_Iterator;

--------------------------------------------------------------------------------


   Function ada_c_Has_More_Elements(d : in designer) return BOOLEAN;

     pragma interface(C,ada_c_Has_More_Elements);
     pragma interface_name(ada_c_Has_More_Elements,
                          "_ada_c_Has_More_Elements__FP16InstanceIterator");

   Function Has_More_Elements(d : in designer) return BOOLEAN is

begin

return ada_c_Has_More_Elements(d);

 end Has_More_Elements;

--------------------------------------------------------------------------------

end designer_PKG;

================================================================================
======
--
-- Component:  db_utility_PKG Spec
-- Author:     Osman Ibrahim
-- Date:       SEP. 1995
-- Language:   Ada
-- Compiler:   caps-suns7 SunAda
-- Purpose:    This package spec represents an Ada mirrored image for some of
--             Ontos Free functions that access and manipulate objects and
--             other DB operations. These operations needs to be extended in
--             the future in the same way as the need arises.

================================================================================
======

with a_strings;
use a_strings;

with designer_PKG;
use  designer_PKG;

package db_utility_PKG is

-- General ONTOS operations
```

```
   procedure open_database(ddb : in a_string);

   procedure close_database(ddb : in a_string);

   procedure transaction_start;

   procedure transaction_commit;

   procedure save_to_db(d  : in designer);

   procedure delete_from_db(d : in designer);

-----------------------------------------------------------------------------

end db_utility_PKG;




------------------------------------------------------------------------------
--
-- Component:  db_utility_PKG Spec
-- Author:     Osman Ibrahim
-- Date:       SEP. 1995
-- Language:   Ada
-- Compiler:   caps-suns7 SunAda
-- Purpose:    This package body represents an Ada mirrored image for some of
--             Ontos Free functions that access and manipulate objects and
--             other DB operations. These operations needs to be extended in
--   the future in the same way as the need arises.
--             Procedure names are given followed by their interface
--             (C in all cases) and interface name.
--
--             Refer to the designer_PKG body for some other detail regarding
--             the Conventions used for subprograms names

--             Refer to the designer PKG Spec for some other detail regarding
--             the way I used to interface Ada to a C++ code.

--             Refer to the file exception_interface.h for the meaning of each
--             exceptiuons used here.
--
-------------------------------------------------------------------------------

with a_strings;
use a_strings;

with c_strings;
use c_strings;

with exception_interface_PKG;
use  exception_interface_PKG;

with designer_PKG; -- needed for type "designer" to be visible here which I do
                   -- not think it is right, it is needed because some functions
```

```
                    -- have "designer" as an input parameter..refer to the note
 -- below.
use  designer_PKG;

package body db_utility_PKG is

--------------------------------------------------------------------------------

   procedure ada_c_open_database(ddb : in c_string) ;

     pragma interface(C,ada_c_open_database);
     pragma interface_name(ada_c_open_database, "_ada_c_open_database__FPc");

--------------------------------------------------------------------------------

   procedure open_database(ddb : in a_string) is

begin

 ada_c_open_database(to_c(ddb));

       case get_exception_code is
               when normal_code                  =>
                      null;
               when db_open_failed_code          =>
                      raise db_open_failed;
               when others                       =>
                      raise Ontos_failure;
       end case;


   end open_database;

--------------------------------------------------------------------------------

   procedure ada_c_close_database(ddb : in c_string);

     pragma interface(C,ada_c_close_database);
     pragma interface_name(ada_c_close_database, "_ada_c_close_database__FPc");

--------------------------------------------------------------------------------


   procedure close_database(ddb : in a_string) is

      begin

          ada_c_close_database(to_c(ddb));

          case get_exception_code is
               when normal_code                  =>
                      null;
               when db_not_open_code             =>
                      raise db_not_open;
               when others                       =>
```

```
                      raise Ontos_failure;
          end case;

      end close_database;

--------------------------------------------------------------------------------


   procedure ada_c_transaction_start;

     pragma interface(C,ada_c_transaction_start);
     pragma interface_name(ada_c_transaction_start,
                           "_ada_c_transaction_start__Fv");

--------------------------------------------------------------------------------

   procedure transaction_start is

begin

ada_c_transaction_start;

         case get_exception_code is
               when normal_code                  =>
                      null;
               when others                       =>
                      raise Ontos_failure;
          end case;

end transaction_start;

--------------------------------------------------------------------------------

   procedure ada_c_transaction_commit;

     pragma interface(C,ada_c_transaction_commit);
     pragma interface_name(ada_c_transaction_commit,
                           "_ada_c_transaction_commit__Fv");

   procedure transaction_commit is

begin

ada_c_transaction_commit;

         case get_exception_code is
               when normal_code                  =>
                      null;
               when no_active_transaction_code   =>
                      raise no_active_transaction;
               when others                       =>
                      raise Ontos_failure;
          end case;
end transaction_commit;

-------------------------------------------------------------------------------

-- I think the following 2 operations should be moved to the designer ADT and
-- renamed "save_designer__to_db" and "delete_designer_from_db" respectively
-- because they depend on the type of object passed and we can not make the
-- input type generic ... can we?????

-------------------------------------------------------------------------------

   procedure ada_c_save_to_db(d  : in designer);

     pragma interface(C,ada_c_save_to_db);
     pragma interface_name(ada_c_save_to_db, "_ada_c_save_to_db__FP6Person");

-------------------------------------------------------------------------------


   procedure save_to_db(d  : in designer) is

begin

ada_c_save_to_db(d);

            case get_exception_code is

               when normal_code                 =>
                      null;
               when object_already_exists_code  =>
                      raise object_already_exists;
               when others                      =>
                      raise Ontos_failure;
           end case;


end save_to_db;

-------------------------------------------------------------------------------

   procedure ada_c_delete_from_db(d : in designer);

     pragma interface(C,ada_c_delete_from_db);
     pragma interface_name(ada_c_delete_from_db,
                           "_ada_c_delete_from_db__FP6Person");

-------------------------------------------------------------------------------


   procedure delete_from_db(d : in designer) is

begin

ada_c_delete_from_db(d);
```

```
            case get_exception_code is

               when normal_code                    =>
                      null;
               when no_such_object_code            =>
                      raise no_such_object;
               when others                         =>
                      raise Ontos_failure;
           end case;

end delete_from_db;

-----------------------------------------------------------------------

end db_utility_PKG;

-------------------------------------------------------------------------------



=======================================================================
======
--
-- Component:  exception_interface_PKG Spec
-- Author:     Osman Ibrahim
-- Date:       SEP. 1995
-- Language:   Ada
-- Compiler:   caps-suns7 SunAda
-- Purpose:    This package spec represents an Ada mirrored image for the
--             coresponding C++ unit "exception_interface". It is made in a
--             separate PKG because most of these exceptions apply to all
--             types. In the future it will be easy to use those exceptions
--             as is or renamed to suit a specific type.

--             One function from the coresponding C++ unit "exception_
--             interface" is missing intentionally here which is :
--             "set_exception_code()" because it is not needed to be visible
--             in the Ada side.

--             Refer to the unit "exception_interface.h" for the meaning
--             of each of the exceptions defined here.
=======================================================================
======

package exception_interface_PKG is

------------------------------------------------------------------------------

-- EXCEPTIONS

   Ontos_failure            :  exception;

   db_open_failed           :  exception;
   db_not_open             :  exception;

   no_active_transaction   :  exception;

   no_such_object          :  exception;

   object_already_exists   :  exception;

---------------------------------------------------------------------------

-- Exception codes captured and returned to ada from ONTOS
-- Refer to the comment in the PKG body.

   type exception_code is (normal_code,
                           object_already_exists_code,
                           no_such_object_code,
                           db_open_failed_code,
                           db_not_open_code,
                           no_active_transaction_code,
                           Ontos_failure_code          );

-------------------------------------------------------------------------------

   Function get_exception_code return exception_code;

------------------------------------------------------------------------------

end exception_interface_PKG;


===============================================================================
======
--
-- Component:  exception_interface_PKG body
-- Author:     Osman Ibrahim
-- Date:       SEP. 1995
-- Language:   Ada
-- Compiler:   caps-suns7 SunAda
-- Purpose:    This package body represents an Ada mirrored image for the
--             coresponding C++ unit "exception_interface". It is made in a
--             separate PKG because most of these exceptions apply to all
--             types. In the future it will be easy to use those exceptions
--             as is or renamed to suit a specific type.

--             One function from the coresponding C++ unit "exception_
--             interface" is missing intentionally here which is :
--             "set_exception_code()" because it is not needed to be visible
--             in the Ada side.

--             Refer to the unit "exception_interface.h" for the meaning
--             of each of the exceptions defined here.
===============================================================================
======
with text_io;
use  text_io;

package body exception_interface_PKG is

-------------------------------------------------------------------------------

-- This function was introduced to allow ada to capture an exception raised
-- inside ONTOS so that Ada can handle it in a way taht will not cause
-- the program to abort because of a user error; e.g a misspelled DB name.

-------------------------------------------------------------------------------

   Function ada_c_get_exception_code return integer;

     pragma interface(C,ada_c_get_exception_code);
     pragma interface_name(ada_c_get_exception_code,
                           "_ada_c_get_exception_code__Fv");

-------------------------------------------------------------------------------

   Function get_exception_code return exception_code is


    begin

      case ada_c_get_exception_code is

            when exception_code'POS(normal_code)                      =>
                  return normal_code;

            when exception_code'POS(no_such_object_code)              =>
                  return no_such_object_code;

            when exception_code'POS(object_already_exists_code)       =>
                  return object_already_exists_code;

            when exception_code'POS(db_open_failed_code)              =>
                  return db_open_failed_code;

            when exception_code'POS(db_not_open_code)                 =>
                  return object_already_exists_code;

            when exception_code'POS(no_active_transaction_code)       =>
                  return no_active_transaction_code;

            when others                                               =>
                  return Ontos_failure_code;

      end case;

    end get_exception_code;

-------------------------------------------------------------------------------
```

```
end exception_interface_PKG;


=====================================================================
======
--
-- Component:  link_with_pragmas_PKG
--
-- Author:     Osman Ibrahim
-- Date:       AUG. 1995
-- Language:   Ada
-- Compiler:   caps-suns7 SunAda
-- Purpose:    This package the pragma link_with for the a.ld pre-link to the
--             the relevant C++ components given by thier object files.
--             It also contains pragma link_with for the a.ld pre-link to the
--             procedure cplusplus_init which is required at TAE level to make
--             TAE, Ada, C++, and ONTOS behave friendlt together !!!!!!
--
--             For linking with a C++ (generally foreign lang.) object file,
--             there are 3 ways to do that :

--                         - use withn  (not recommended by Ada documentations)

--                         - supply the C++ (.o) files  in the command
--                           line as opetions to a.ld , this is doable and
--                            can be included in the Makefile

--                         - The way I'm using is to use link_with pragma to
--                            link to the desired (.o) file

--             Refer to the files designer_s.a and designer_b.a for other
--             relevant details of how interfacing Ada to C++

=====================================================================
======

package link_with_pragmas_PKG is

-- OSMAN 7/24
-- CAPS C++ operation to initialize static constructors

   procedure cplusplus_init;
     pragma interface(C,cplusplus_init);
     pragma interface_name(cplusplus_init, "_main");

--------------------------------------------------------------------------------

-- Main C++ interface object module, required for initialization of C++
-- static constructors.

   pragma link_with("C++_initialize.o");

   pragma link_with("person.o");
   pragma link_with("person_interface.o");

   pragma link_with("db_utility.o");

   pragma link_with("exception_interface.o");


end link_with_pragmas_PKG;
-- OSMAN




=====================================================================
======

-- Unit             :  designer_ops.a
-- Purpose          :  Interface experiment
-- Date             :  Jul 26, 95 and modified Oct 95
-- Author           :  Osman Ibrahim
-- Compiler         :  SunAda
-- Description      :  This module has originally appeared in ECS under the same
--                     name (designer_ops.a) and was coded completely in C++
--                     I translated it into Ada to examine the possibilty of
--                     directly using C++ classes from inside ADA and thus
--                     testing the new approach. It should be noted that without
--                     the new approach of interfacing Ada to C++, this module
--                     could not be coded in Ada
--                     This Ada module provide the same functionality that is
--                     currently provided by the coresponding C++ module for
--                     the designer pool in the ECS.
--                     I did not try to change any logic or implementation here
--                     to test the new approach.

--                     Notice that some of the code here is redundant and is not
--                     needed especialy the checks to see if the designer already
--                     exists in the DB (or the parallel check to see if the
--                     designer does not exists in the DB) before performing some
--                     operations. The lower level operations defined in the
--                     designer and db_utility packages guards aginst the
--                     occurences of such conditions by raising the proper
--                     exception.

=====================================================================
======

with link_with_pragmas_PKG;
use link_with_pragmas_PKG;

with designer_PKG;
use designer_PKG;

with db_utility_PKG;
use  db_utility_PKG;
```

```
with exception_interface_PKG;
use  exception_interface_PKG;

with c_strings;
use c_strings;

with a_strings;
use a_strings;

with u_env;
use u_env;

with text_io;
use text_io;

----------------------------------------------------------------------------


package Designer_Ops_PKG is


 package int_io is new integer_io(integer);
 use  int_io;

  procedure create_designer (name : in a_string; level :in expertise_level);

  procedure write_designers_to_file;

  procedure add_designer (name : in a_string; level :in expertise_level);

  procedure delete_designer (name : in a_string);

  procedure change_exp_level (name : in a_string; level :in expertise_level);

  procedure change_status(name : in a_string);

  procedure show_designer(name : in a_string);

  procedure show_all_designers;

end Designer_Ops_PKG;

----------------------------------------------------------------------------

package body Designer_Ops_PKG is

----------------------------------------------------------------------------

-- This procedure iterates through the designer instances (using the new
-- iterator) and write designer info into a file called:
-- "/.caps/temp/ddbdisplay" to be used later by TAE to display this info
-- in the designer panel. Notice the use of the new ITERATOR here.
```

```
procedure write_designers_to_file is

local_designer      : designer;
designer_iterator   : designer;

 user_directory: a_string :=
       copy(c_strings.to_a(u_env.getenv(c_strings.to_c("HOME"))) &
       a_strings.to_a("/.caps/temp/ddbdisplay")) ;
 ECS_output : file_type;
 ECS_output_file_name : a_strings.a_string := user_directory;
 begin

      transaction_start;

      open(ECS_output, MODE => OUT_FILE, NAME => ECS_output_file_name.s);

      designer_iterator := Create_Instance_Iterator(to_a("Person"));

      while ( Has_More_Elements(designer_iterator)) loop

           local_designer := Get_Next_Element(designer_iterator);

           put(ECS_output,GetPersonName(local_designer).s);
           SET_COL(ECS_output,25);
           if GetPersonLevel(local_designer) = low then
              put(ECS_output, "Low");
           elsif GetPersonLevel(local_designer) = medium then
              put(ECS_output, "Med");
           else
              put(ECS_output, "High");
           end if;

           SET_COL(ECS_output,45);
           if GetPersonStatus(local_designer) = free then
              put(ECS_output, "Free");
           else
              put(ECS_output, "Busy");
           end if;

           NEW_LINE(ECS_output);
       end loop;
       close(ECS_output);
       transaction_commit;

    end write_designers_to_file;
--------------------------------------------------------------------------------

procedure create_designer (name : in a_string;
                           level :in expertise_level ) is

local_designer : designer;

 begin
```

```
     local_designer := new_designer(name, level, free);

    end create_designer;

-------------------------------------------------------------------------------

-- Add designer to the DB
-- Note that when adding a new designer to the DB his status is free by default
-- that is the reason we do not need status as a parameter.

  procedure add_designer (name : in a_string;
                          level :in expertise_level ) is

  local_designer : designer;

   begin

      transaction_start;
      local_designer := get_designer(name);
      if local_designer /= null then
         PUT_LINE("Designer already exists in the DB");
         null;
      else
         local_designer:= new_designer(name, level, free);
         save_to_db(local_designer);
      end if ;
      transaction_commit;

   end add_designer;

-------------------------------------------------------------------------------


  procedure delete_designer (name : in a_string) is

  local_designer : designer;

   begin

      transaction_start;
      local_designer := get_designer(name);
      if local_designer = null then
         PUT_LINE("Designer does not exist in the DB");
       null;
      else
         delete_from_db(local_designer);
      end if ;
      transaction_commit;

   end delete_designer;

-------------------------------------------------------------------------------

   procedure change_exp_level (name : in a_string;
                                          level: in expertise_level) is

  local_designer : designer;

    begin

      transaction_start;
      local_designer := get_designer(name);
      if local_designer = null then
         PUT_LINE("Designer Does not exist in the DB");
        null;
      else
         SetPersonLevel(level, local_designer);
         save_to_db(local_designer);
      end if ;
      transaction_commit;

    end change_exp_level;

--------------------------------------------------------------------------------

  procedure change_status(name : in a_string) is

  local_designer : designer;

    begin
      transaction_start;
      local_designer := get_designer(name);
      if local_designer = null then
         PUT_LINE("Designer does not exist in the DB");
        null;
      else
         if GetPersonStatus(local_designer) = free  then
            SetPersonStatus(busy, local_designer);
         else
            SetPersonStatus(free, local_designer);
         end if ;
         save_to_db(local_designer);
      end if ;
      transaction_commit;

    end change_status;

--------------------------------------------------------------------------------

procedure show_designer(name: in a_string) is

  local_designer : designer;

   begin
      transaction_start;
      local_designer := get_designer(name);
      if local_designer = null then
         PUT_LINE("Designer does not exist in the DB");
       null;
      else
         PutDesigner(local_designer);
      end if ;
      transaction_commit;

   end show_designer;

------------------------------------------------------------------------------

procedure show_all_designers is

 local_designer      : designer;
 designer_iterator   : designer;

  begin

    transaction_start;
    designer_iterator := Create_Instance_Iterator(to_a("Person")) ;
    while ( Has_More_Elements(designer_iterator)) loop
    local_designer := Get_Next_Element(designer_iterator);
    if local_designer = null then
         PUT_LINE("Designer does not exist in the DB");
      null;
     else
         PutDesigner(local_designer);
      end if ;

  end loop;
  transaction_commit;

  end show_all_designers;

------------------------------------------------------------------------------

end Designer_Ops_PKG;

------------------------------------------------------------------------------

==============================================================================
======
--
-- CAPS (Version 3) Edit Designer Team Operations Interface
--
-- Date:             Aug 1995
--
-- Author:           Osman Ibrahim
-- Compiler:         SunAda
--
--
-- This interface provides access to ECS designer pool modification
-- procedures.
--
-- This code was generated using TAE 5.3 and modified to integrate the
-- CAPS tools. The code was adapted from jim original code so that it
-- can be tested in isolation away from other ECS components. But it provides
-- the same original functionality. Among other changes is incorprating of some
-- (triveal) exception handlers to test the working of capturing and handling
-- exceptions raised inside ONTOS. These exception handler should be elaborated
-- and enhanced in a way that makes excution resumes in the correct path (if
-- recoverable) through a user friendly interface using TAE like showing a pnnel
-- where the error will be explained and allow the user to correct it in that
-- panel (if possible) and then resume the excution.
--
-- Modifications:Oct 1995
--
--
==============================================================
======

with tae;             use tae;
with X_Windows;
with text_io;         use text_io;
with a_strings;       use a_strings;
with c_strings;       use c_strings;
with u_env;           use u_env;
with system;          use system;

--with ecs_operations;        --use ecs_operations;
with CAPS_additional_TAE;   use CAPS_additional_TAE;
with link_with_pragmas_PKG;      use link_with_pragmas_PKG;
with Designer_Ops_PKG;                 use Designer_Ops_PKG;
with designer_PKG;          use designer_PKG;
with CAPS_alert_package;    use CAPS_alert_package;
with db_utility_PKG;        use db_utility_PKG;
with exception_interface_PKG;
use  exception_interface_PKG;

procedure test_edit_team is

package edit_team_support is

    package taefloat_io is new text_io.float_io (taefloat);
    package taeint_io is new text_io.integer_io(taeint);
    package int_io is new text_io.integer_io(integer); use int_io;
    procedure initializePanels (file : in string);  -- NOTE: params changed
    procedure sort_designer_file;

    -- BEGIN EVENT_HANDLERs
    procedure editteam_name (info : in tae_wpt.event_context_ptr);
    procedure editteam_ex_opt (info : in tae_wpt.event_context_ptr);
    procedure editteam_d_cancel (info : in tae_wpt.event_context_ptr);
    procedure editteam_designers (info : in tae_wpt.event_context_ptr);
    procedure editteam_selection_3 (info : in tae_wpt.event_context_ptr);
    procedure confirm_yes (info : in tae_wpt.event_context_ptr);
    procedure confirm_no (info : in tae_wpt.event_context_ptr);
    -- END EVENT_HANDLERs

end edit_team_support;
```

```
----------------------------------------------------------

    use edit_team_support;
    use tae.tae_misc;

    theDisplay : X_Windows.Display;
    user_ptr : tae_wpt.event_context_ptr;
    editteam_info : tae_wpt.event_context_ptr;
    confirm_info : tae_wpt.event_context_ptr;
    etype : wpt_eventtype;
    wptEvent : tae_wpt.wpt_eventptr;

    dummy : boolean;   -- used to clear out the wpt event queue

    time_to_exit : exception;

    MAX_DESIGNERS     : integer := 20;

    designer_info     : s_vector(1..MAX_DESIGNERS) := (others => new
string(1..64));
    designer          : a_string := null;  -- changed from c_string to a_string
    expertise_level   : string(1..24);
    designer_status   : string(1..24);
    expertise_level_code : designer_PKG.expertise_level := low;

    data_file         : text_io.file_type;
    file_name         : a_string := null;
    length            : integer;
    counter           : integer;

    null_string       : string(1..64) := (others => ascii.nul);
    temp_string       : string(1..64) := null_string;
    test_string       : string(1..64);
    time_string       : string(1..14);
    c_time_string     : c_string := null;
    designer_assigned: boolean := false;
    priority          : integer;
    step_set_cardinality : integer := 0;

    open_prototype_file      : file_type;
    open_prototype_file_name : a_string := null;

    user                     : a_string := null;
    user_home_dir_name       : a_string := null;
     user_home               : a_string := null;
    ddb_name                 : a_string := to_a("hoda_db");
                                  -- changed from c_string to a_string
    caps_home                : a_string := null;


----------------------------------------------------------


package body edit_team_support is
```

```
procedure initializePanels (file : in string) is

use tae.tae_co;
use tae.tae_misc;

tmp_info : tae_wpt.event_context_ptr;
   dummy    : BOOLEAN;

    begin

-- do one Co_New and Co_ReadFile per resource file
tmp_info := new tae_wpt.event_context;
Co_New (0, tmp_info.collection);
-- could pass P_ABORT if you prefer
Co_ReadFile (tmp_info.collection, file, P_CONT);

       -- pair of Co_Finds for each panel in this resource file

        editteam_info := new tae_wpt.event_context;
        editteam_info.collection := tmp_info.collection;
        Co_Find (editteam_info.collection, "editteam_v", editteam_info.view);
        Co_Find (editteam_info.collection, "editteam_t", editteam_info.target);

        confirm_info := new tae_wpt.event_context;
        confirm_info.collection := tmp_info.collection;
        Co_Find (confirm_info.collection, "confirm_v", confirm_info.view);
        Co_Find (confirm_info.collection, "confirm_t", confirm_info.target);

       -- Since there can now be MULTIPLE INITIAL PANELS defined from
       -- within the TAE WorkBench, call Wpt_NewPanel for each panel
       -- defined to be an initial panel (but not usually all the panels
       -- which appear in the resource file).


        if editteam_info.panel_id = NULL_PANEL_ID then
            tae_wpt.Wpt_NewPanel (theDisplay, editteam_info.target,
editteam_info.view,
               X_Windows.Null_Window, editteam_info, tae_wpt.WPT_PREFERRED,
               editteam_info.panel_id);
        else
            tae_wpt.Wpt_SetPanelState (
               editteam_info.panel_id, tae_wpt.WPT_PREFERRED);
        end if;

-- osman

       -- Get user name and home directory.

       user := c_strings.to_a(u_env.getenv(c_strings.to_c("USER")));
       user_home := c_strings.to_a(u_env.getenv(c_strings.to_c("HOME")));

       -- read the designer pool and put it in the panel
       write_designers_to_file; -- get designers from DDB and put in
                                -- "$.caps/temp/ddbdisplay"
       -- read in the designers from the transfer file (ddbdisplay)
       -- into the editteam panel

       sort_designer_file;
       text_io.open(data_file, text_io.IN_FILE,user_home.s&"/.caps/temp/ddbdis-
play");
       counter := 1;
       while not end_of_file(data_file) loop

         get_line(data_file,designer_info(counter).all,length);
         if length > 64 then
            put_line(" length > 64");
         end if;
         TAE_Wpt.Wpt_SetStringConstraints(editteam_info.panel_id,
                  "designers", TaeInt(counter), designer_info);
         counter := counter + 1;

       end loop;

       text_io.CLOSE(data_file);

       for i in counter..MAX_DESIGNERS loop
         designer_info(i).all :=
         "                                                                ";
         TAE_Wpt.Wpt_SetStringConstraints(editteam_info.panel_id,
                  "designers", TaeInt(counter), designer_info);

       end loop;
-- osman

       dummy := Tae_Wpt.Wpt_Pending;

     end initializePanels;

procedure sort_designer_file is

  MAX_DESIGNERS   : integer := 500;
  null_string     : string(1..64) := (others => ' ');  -- Using ' ' rather than
ascii.nul because when
                                                        -- the TAE application reads
this file,it stops
                                                        -- reading when it sees null
characters.
  designer_array : array(1..MAX_DESIGNERS) of string(1..64) := (others =>
null_string);
  designer_file   : file_type;
  counter         : integer := 1;
  length          : integer := 0;
  designer        : string(1..64) := null_string;
  temp_string     : string(1..64) := null_string;
  inserted        : boolean := false;

begin

  open(designer_file,in_file,user_home.s&"/.caps/temp/ddbdisplay");
  while not end_of_file(designer_file) loop
    designer := null_string;
    get_line(designer_file,designer,length);
    inserted := false;
    for j in 1..counter-1 loop  -- check all previously inserted designers
      if designer < designer_array(j) then -- if we found the insertion point
        for k in reverse j+1..counter loop
          if (k>1) then
             designer_array(k) := designer_array(k-1); end if; -- move all
"greater than"
                                                               -- designers up 1
        end loop;
        designer_array(j) := designer;  -- insert this designer
        inserted := true;
        exit;  -- exit for j loop
      end if;
    end loop;
    if not inserted then designer_array(counter) := designer; end if;
             -- add designer to end of array because it
             -- is lexicographically greater than the others
    counter := counter + 1;
  end loop;
  delete(designer_file);

  create(designer_file,out_file,user_home.s&"/.caps/temp/ddbdisplay");
  for i in 1..counter-1 loop
    put_line(designer_file,trim(to_a(designer_array(i))).s);
  end loop;
  close(designer_file);

end sort_designer_file;

--
--BEGIN EVENT_HANDLERs
--



procedure editteam_name (info : in tae_wpt.event_context_ptr) is
    value : array (1..1) of string (1..tae_taeconf.STRINGSIZE);
    count : taeint;

        begin
        tae_vm.Vm_Extract_Count (info.parm_ptr, count);
        if count <= 0 then null;
        else tae_vm.Vm_Extract_SVAL (info.parm_ptr, 1, value(1));
        end if;
        designer_assigned := true;
        designer := trim(to_a(value(1)));
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"expertise",
                        "       ");
        expertise_level_code := low;  -- reset default to low
        expertise_level(1..3) := "low";
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"status",
                        "         ");
        designer_status(1..4) := "    ";

    end editteam_name;

procedure editteam_ex_opt (info : in tae_wpt.event_context_ptr) is
    value : array (1..1) of string (1..tae_taeconf.STRINGSIZE);
    count : taeint;

        begin
        tae_vm.Vm_Extract_Count (info.parm_ptr, count);
        if count <= 0 then null;
        else tae_vm.Vm_Extract_SVAL (info.parm_ptr, 1, value(1));
        end if;
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"expertise",value(1));
        expertise_level(1..24) := value(1)(1..24);
    end editteam_ex_opt;

procedure editteam_d_cancel (info : in tae_wpt.event_context_ptr) is
    value : array (1..1) of string (1..tae_taeconf.STRINGSIZE);
    count : taeint;

        begin
        tae_vm.Vm_Extract_Count (info.parm_ptr, count);
        if count <= 0 then null;
        else tae_vm.Vm_Extract_SVAL (info.parm_ptr, 1, value(1));
        end if;
        if info.panel_id = NULL_PANEL_ID then
            tae_wpt.Wpt_NewPanel (theDisplay, info.target, info.view,
               X_Windows.Null_Window, info, tae_wpt.WPT_PREFERRED,
               info.panel_id);
        else
            tae_wpt.Wpt_SetPanelState (
               info.panel_id, tae_wpt.WPT_PREFERRED);
        end if;

        designer_assigned := false;
        designer := null;
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"name",
                        "                                ");
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"expertise",
                        "        ");
        expertise_level_code := low;  -- reset default to low
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"status",
                        "           ");
        designer_status(1..4) := "    ";

    end editteam_d_cancel;

procedure editteam_designers (info : in tae_wpt.event_context_ptr) is
    value : array (1..1) of string (1..tae_taeconf.STRINGSIZE);
    count : taeint;

        begin
        tae_vm.Vm_Extract_Count (info.parm_ptr, count);
        if count <= 0 then null;
        else tae_vm.Vm_Extract_SVAL (info.parm_ptr, 1, value(1));
        end if;
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"name",
                        value(1)(1..24));
        designer_assigned := true;
        designer := trim(to_a(value(1)(1..24)));
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"expertise",
                        value(1)(25..30));
        expertise_level(1..6) := value(1)(25..30);
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"status",
                        value(1)(44..51));
        designer_status(1..4) := value(1)(44..47);

    end editteam_designers;

procedure editteam_selection_3 (info : in tae_wpt.event_context_ptr) is
    value : array (1..1) of string (1..tae_taeconf.STRINGSIZE);
    count : taeint;

        begin

        tae_vm.Vm_Extract_Count (info.parm_ptr, count);
        if count <= 0 then null;
        else tae_vm.Vm_Extract_SVAL (info.parm_ptr, 1, value(1));
        end if;
        if (FALSE) then null;
        elsif s_equal (value(1), "add designer") then

        -- add designer to ddb

          if not designer_assigned then designer := null;
                 caps_alert(to_a("ERROR: No designer selected."));
               write_designers_to_file;

          else

            expertise_level_code := low; -- default to low expertise level
            if expertise_level(1..3) = "low" or expertise_level(1..3) = "Low"
                                        then expertise_level_code := low;
            end if;
            if expertise_level(1..3) = "med" or expertise_level(1..3) = "Med"
                                        then expertise_level_code := medium;
            end if;
            if expertise_level(1..2) = "hi"  or expertise_level(1..3) = "Hi"
                                        then expertise_level_code := high;
            end if;

            -- add designer and write new designer list to $HOME/ddbdisplay

            add_designer(designer,expertise_level_code);

         end if;

          caps_alert(to_a("Designer addition complete."));
          -- now read the new transfer file and update the TAE item and

        write_designers_to_file;   -- osman new
          sort_designer_file;
          text_io.open(data_file, text_io.IN_FILE,user_home.s&"/.caps/temp/ddb-
display");
          counter := 1;

          while not end_of_file(data_file) loop
            get_line(data_file,designer_info(counter).all,length);
            TAE_Wpt.Wpt_SetStringConstraints(editteam_info.panel_id,
                       "designers", TaeInt(counter), designer_info);
            counter := counter + 1;
          end loop;

          text_io.CLOSE(data_file);

          for i in counter..MAX_DESIGNERS loop
            designer_info(i).all :=
            "                                                           ";
            TAE_Wpt.Wpt_SetStringConstraints(editteam_info.panel_id,
                      "designers", TaeInt(counter), designer_info);
          end loop;


        elsif s_equal (value(1), "delete designer") then
            if confirm_info.panel_id = NULL_PANEL_ID then
                tae_wpt.Wpt_NewPanel (theDisplay, confirm_info.target,
confirm_info.view,
                   X_Windows.Null_Window, confirm_info, tae_wpt.WPT_PREFERRED,
                   confirm_info.panel_id);
            else
                tae_wpt.Wpt_SetPanelState (
                   confirm_info.panel_id, tae_wpt.WPT_PREFERRED);
            end if;

        elsif s_equal (value(1), "change expertise level") then

        -- update designer info in ddb

          begin

            if not designer_assigned then designer := null;
                         caps_alert(to_a("ERROR: No designer selected."));
                         write_designers_to_file;
            else

              if expertise_level(1..3) = "low" or expertise_level(1..3) = "Low"
                 then expertise_level_code := low; end if;
              if expertise_level(1..3) = "med" or expertise_level(1..3) = "Med"
                 then expertise_level_code := medium; end if;
              if expertise_level(1..3) = "hig" or expertise_level(1..3) = "Hig"
                 then expertise_level_code := high; end if;

              change_exp_level(designer,expertise_level_code);
```

```
                write_designers_to_file;    -- osman new


             end if;

             caps_alert(to_a("Designer expertise modification complete."));

             -- now read the new transfer file and update the TAE item

             sort_designer_file;
            text_io.open(data_file, text_io.IN_FILE,user_home.s&"/.caps/temp/ddb-
display");
             counter := 1;

             while not end_of_file(data_file) loop

                  get_line(data_file,designer_info(counter).all,length);
                  TAE_Wpt.Wpt_SetStringConstraints(editteam_info.panel_id,
                          "designers", TaeInt(counter), designer_info);
                  counter := counter + 1;

             end loop;

             text_io.CLOSE(data_file);

             for i in counter..MAX_DESIGNERS loop

               designer_info(i).all :=
                 "                                                          ";
               TAE_Wpt.Wpt_SetStringConstraints(editteam_info.panel_id,
                         "designers", TaeInt(counter), designer_info);

             end loop;

           end;

         elsif s_equal (value(1), "return to main CAPS menu") then

             tae_wpt.Wpt_PanelReset(editteam_info.panel_id);

             if not (editteam_info.panel_id = NULL_PANEL_ID) then
                tae_wpt.Wpt_PanelErase(info.panel_id);   end if;
             raise time_to_exit;

         end if;

     end editteam_selection_3;

procedure confirm_yes (info : in tae_wpt.event_context_ptr) is
    value : array (1..1) of string (1..tae_taeconf.STRINGSIZE);
    count : taeint;

        begin
        tae_vm.Vm_Extract_Count (info.parm_ptr, count);
        if count <= 0 then null;
        else tae_vm.Vm_Extract_SVAL (info.parm_ptr, 1, value(1));
        end if;

        -- remove designer from ddb

        if not designer_assigned then designer := null;
              caps_alert(to_a("ERROR: No designer selected."));
              write_designers_to_file;
        else
          delete_designer(designer);

           write_designers_to_file;        -- osman new

           if designer_status(1..4) = "Busy" then
                caps_alert(to_a
  ("NOTICE:  The designer just deleted was busy, RESCHEDULING his/her tasks."));
           end if;

        end if;

        tae_wpt.Wpt_PanelErase(info.panel_id);
        tae_wpt.Wpt_PanelReset(editteam_info.panel_id);
        --caps_alert(to_a("Designer deletion complete."));


      TAE_TERMIO.T_BELL ;
      TAE_WPT.WPT_MessageNoBlock(editteam_info.panel_id,
                    "Designer deletion complete.");



        -- clear the TAE panel items

        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"name",
                         "                              ");
        designer := null;
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"expertise",
                         "       ");
        expertise_level_code := low;
        Tae_Wpt.Wpt_SetString(editteam_info.panel_id,"status",
                         "         ");

        -- read the new designer list from the transfer file

        sort_designer_file;
        text_io.open(data_file, text_io.IN_FILE,user_home.s&"/.caps/temp/ddbdis-
play");
        counter := 1;

        while not end_of_file(data_file) loop

           get_line(data_file,designer_info(counter).all,length);
           TAE_Wpt.Wpt_SetStringConstraints(editteam_info.panel_id,
                    "designers", TaeInt(counter), designer_info);
           counter := counter + 1;
        end loop;

        text_io.CLOSE(data_file);

        for i in counter..MAX_DESIGNERS loop

           designer_info(i).all :=
             "                                                         ";
           TAE_Wpt.Wpt_SetStringConstraints(editteam_info.panel_id,
                    "designers", TaeInt(counter), designer_info);

        end loop;

     end confirm_yes;


procedure confirm_no (info : in tae_wpt.event_context_ptr) is

        begin

        caps_alert(to_a("Cancelling designer deletion."));

        -- do nothing
        tae_wpt.Wpt_PanelErase(info.panel_id);

    end confirm_no;

--END EVENT_HANDLERs


end edit_team_support;

--------------------------------------------------------------------------

--
-- Main Program
--

  begin

--tae_wpt.Wpt_Init ("",theDisplay);

-- Note that we are using the specially designed Wpt_CCInit procedure. This
-- is so that we, rather that TAE, initialize all C++ static constructors.
    cplusplus_init; -- initialize C++ static constructors for ONTOS and TAE
    f_force_lower (FALSE);    -- permit upper/lowercase file names
    CAPS_additional_TAE.Wpt_CCInit("",theDisplay);
    tae_wpt.Wpt_NewEvent (wptEvent);
    caps_home := c_strings.to_a(u_env.getenv(c_strings.to_c("CAPSHOME")));
    open_database(ddb_name);  -- the coresponding C++ function is coded in a

   transaction_commit;  -- for testing the exception no_active transaction

    initializePanels ( "osman_edit_team.res");   -- single call
```

```
--      main event loop

EVENT_LOOP:
    loop
        tae_wpt.Wpt_NextEvent (wptEvent, etype);     -- get next event

-- NOTE: This case statement includes STUBs for non-WPT_PARM_EVENT events.

        case etype is

            when wpt_eventtype'first .. -1 => null;
                -- iterate loop on Wpt_NextEvent error

-- TYPICAL CASE: Panel Event (WPT_PARM_EVENT)

            when tae_wpt.WPT_PARM_EVENT =>
            -- You can comment out the following "put" call.
            -- The appropriate EVENT_HANDLER finishes the message.
            -- text_io.put ( "Event: WPT_PARM_EVENT, " );

            --      Panel event has occurred.
            --      Get parm name and then call appropriate EVENT_HANDLER.
            --
            --                          CAUTION:
            --      DO NOT call Wpt_Extract_Parm_xEvent from any other branch
            --      of this "case" statement or you'll get "storage_error".
            --
                    tae_wpt.Wpt_Extract_Context (wptEvent, user_ptr);
                    tae_wpt.Wpt_Extract_Parm (wptEvent, user_ptr.parm_name);
                    tae_wpt.Wpt_Extract_Data (wptEvent, user_ptr.datavm_ptr);
                    tae_vm.Vm_Find (user_ptr.datavm_ptr, user_ptr.parm_name,
                                    user_ptr.parm_ptr);

            -- dummy if to ease code generation
                if (FALSE) then null;

                    -- WPT_PARM_EVENT, BEGIN panel editteam

                    elsif tae_wpt."=" (user_ptr, editteam_info)  then
                    if (FALSE) then null;     -- another dummy if
                        -- determine appropriate EVENT_HANDLER for this item
                        elsif s_equal ("name", user_ptr.parm_name) then
                            editteam_name (user_ptr);
                        elsif s_equal ("ex_opt", user_ptr.parm_name) then
                            editteam_ex_opt (user_ptr);
                        elsif s_equal ("d_cancel", user_ptr.parm_name) then
                            editteam_d_cancel (user_ptr);
                        elsif s_equal ("designers", user_ptr.parm_name) then
                            editteam_designers (user_ptr);
                        elsif s_equal ("selection_3", user_ptr.parm_name) then
                            editteam_selection_3 (user_ptr);
                    end if;    -- END panel editteam

                    -- WPT_PARM_EVENT, BEGIN panel confirm
                    elsif tae_wpt."=" (user_ptr, confirm_info)  then
                    if (FALSE) then null;    -- another dummy if
                        -- determine appropriate EVENT_HANDLER for this item
                        elsif s_equal ("yes", user_ptr.parm_name) then
                            confirm_yes (user_ptr);
                        elsif s_equal ("no", user_ptr.parm_name) then
                            confirm_no (user_ptr);
                    end if;    -- END panel confirm

                    else
                        text_io.put_line ("unexpected event from wpt!");
                       exit; -- or raise an exception, but compiler warns if no
exit
                end if;

            when tae_wpt.WPT_FILE_EVENT =>
                        text_io.put_line ("STUB: Event WPT_FILE_EVENT");

                        -- Use Wpt_AddEvent and Wpt_RemoveEvent and
                        -- Wpt_Extract_EventSource and Wpt_Extract_EventMask

            when tae_wpt.WPT_TIMEOUT_EVENT =>
                        text_io.put_line ("STUB: Event WPT_TIMEOUT_EVENT");

                        -- Use Wpt_SetTimeOut for this

-- LEAST LIKELY cases follow:

            when tae_wpt.WPT_WINDOW_EVENT => null ;

                       -- WPT_WINDOW_EVENT can be caused by user acknowledgement
                        -- of a Wpt_PanelMessage or windows which you
                        -- directly create with X (not TAE panels).
                        -- You MIGHT want to use Wpt_Extract_xEvent_Type here.
                        --
                        -- DO NOT use Wpt_Extract_Parm_xEvent since this is not
                        -- a WPT_PARM_EVENT; you'll get a "storage error".

            when tae_wpt.WPT_HELP_EVENT =>          -- OR null ;
                        text_io.put("ERROR: WPT_HELP_EVENT: ");
                      text_io.put_line("should never see; reserved for TAE use");

            when tae_wpt.WPT_INTERRUPT_EVENT =>     -- OR null ;
                        text_io.put("ERROR: WPT_INTERRUPT_EVENT: ");
                      text_io.put_line("should never see; reserved for TAE use");

            when OTHERS =>
                        text_io.put ("FATAL ERROR: Unknown Wpt_NextEvent Event
Type: ");
                        text_io.put (wpt_eventtype'image(etype) ) ;
                        text_io.put_line (" ... Forcing exit.");
                        exit;  -- or raise an exception


        end case;    -- NOTE: Do not add statements between here and "end loop
```

```
    end loop EVENT_LOOP;

    exception

          when object_already_exists =>
               put_line("Error: ");
               put_line("The Designer you Adding Already Exists in DB");

          when no_such_object               =>
               put_line("Error: ");
               put_line("You are retrieving a Designer that does not Exists in
DB");

          when db_open_failed               =>
               put_line("Error: ");
               put_line("Probably you have a wrong DB name or DB is not regis-
tered");

          when db_not_open                  =>
               put_line("Error: ");
               put_line("You are closing a DB that was not Open");

          when no_active_transaction        =>
               put_line("Error: ");
               put_line("You are commiting a Transaction which has not been
started");

           when Ontos_failure               =>
               put_line("Error: ");
               put_line("Ontos Failure");

          when time_to_exit               =>
               put_line("Quitting edit team tool.");
               close_database(ddb_name); -- OSMAN

end test_edit_team;

=============================================================
======
```

APPENDIX B. EXTENDED TEMPLATE CODE

A. A CLASS IMPLEMENTED USING THE RELATION TEMPLATE

1. (.h File)

```
#ifndef __Person_OBJECT_H
#define __Person_OBJECT_H
#ifndef __subordinates_relation_OBJECT_H
#include "subordinates_relation_OBJECT.h"
#endif
#ifndef __supervisors_relation_OBJECT_H
#include "supervisors_relation_OBJECT.h"
#endif
#include <Object.h>
//#include <Reference.h>
//#include <Set.h>

class Person_ENTITY : public subordinates_relation_ENTITY, public
supervisors_relation_ENTITY {
 private:
  int      priv_level; // The designer expertise level

                                                // 0 : low

                                                // 1 : Medium

                                                // 2 : high

  int      priv_status; // The availability status of a designer

                                                // 0 : free

                                                // 1 : busy

 public:

  // Constructors
  Person_ENTITY(char* name=(char*)0,int level= 0, int status=0);

  Person_ENTITY (APL*);     // (Ontos required Constructor)

  // Ontos required member function)
```

```
    virtual Type *getDirectType();

// ---------------------------------------------------------------------

    // For a class to be derived from multiple base classes (multiple
    // inheritance) as our case, ONTOS requires that the following
    // operaations be reimplemented :

    void* operator new(OC_size_t sz);

    void* operator new(OC_size_t sz, APL* theAPL);

    void* operator new(OC_size_t sz, StorageManager* sm, Type* t);

    void  operator delete(void* v);

    virtual void* startAddress() {return this;}

    // ONTOS method for savig Object as pesistent Object.
    virtual void putObject(OC_Boolean deallocate=FALSE);

    // ONTOS method for deleting an Object
    virtual void deleteObject(OC_Boolean deallocate=FALSE);
// ---------------------------------------------------------------------

    // Accessors
    int    GetPersonLevel() ;

    void    SetPersonLevel(int level);

    int    GetPersonStatus();

    void    SetPersonStatus(int status);

    char*   GetPersonName();

    void    SetPersonName(char* name);


// ---------------------------------------------------------------------
};
#endif
```

2. (.cxx File)

```
/* -----------------------------------------------------------------------

-- Unit        : class Person implementation (.cxx)
-- File        : personc.cxx
-- Date        : Documented Oct 5,1995.
-- Author      : Osman Ibrahim
-- Systems     : Sun C++ and ONTOS (2.1)
-- Description : Provides the implementation (definition) for the Person
                 Class that implements the designer ADT in C++
----------------------------------------------------------------------- */
#ifndef __Person_OBJECT_H
#include "Person_OBJECT.h"
#endif
#include <Directory.h>
#include <Set.h>

// Set* Big_Set; //= new Set((Type*)OC_lookup("Person_ENTITY"));


//---------------------------------------------
// constructors
//---------------------------------------------

/* --------------------------------------------------------------------- */

Person_ENTITY::Person_ENTITY(APL* theAPL)
        : subordinates_relation_ENTITY(theAPL),
          supervisors_relation_ENTITY(theAPL)
//          subordinates_relation_ENTITY(theAPL)
/* --------------------------------------------------------------------- */

Person_ENTITY::Person_ENTITY(char* name,int level, int status)
  : subordinates_relation_ENTITY(name)
 {
 subordinates_relation_ENTITY :: initDirectType((Type
*)OC_lookup("Person_ENTITY"));

//  subordinates_relation_ENTITY :: directType( getDirectType( ));
// Name(name);
 priv_level = level;
 priv_status = status;
```

```
}
//------------------------------------------------------------------

  // For a class to be derived from multiple base classes (multiple
  // inheritance) as our case, ONTOS requires that the following
  // operaations be reimplemented :

 void* Person_ENTITY :: operator new(OC_size_t sz)

{
 return subordinates_relation_ENTITY :: operator new(sz);
}

//------------------------------------------------------------------

void* Person_ENTITY :: operator new(OC_size_t sz, APL* theAPL)
{
 return subordinates_relation_ENTITY :: operator new(sz, theAPL);
}

//------------------------------------------------------------------

void* Person_ENTITY :: operator new(OC_size_t sz, StorageManager* sm, Type* t)

{
 return subordinates_relation_ENTITY :: operator new(sz, sm,
                                  (Type*)OC_lookup("Person_ENTITY"));
}
//------------------------------------------------------------------

void  Person_ENTITY :: operator delete(void* v)
{
 subordinates_relation_ENTITY :: operator delete(v);
}

//------------------------------------------------------------------

// ONTOS method for savig Object as pesistent Object.
void Person_ENTITY :: putObject(OC_Boolean deallocate)
{
 subordinates_relation_ENTITY :: putObject(FALSE);
 supervisors_relation_ENTITY  :: putObject(FALSE);
//  subordinates_relation_ENTITY :: putObject(FALSE);
```

```
 if (deallocate) delete this;
}
// -----------------------------------------------------------------
// ONTOS method for deleting an Object
void Person_ENTITY :: deleteObject(OC_Boolean deallocate)
{
 supervisors_relation_ENTITY  :: deleteObject(FALSE);
 subordinates_relation_ENTITY :: deleteObject(FALSE);
 if (deallocate) delete this;
}
// -----------------------------------------------------------------
// Ontos required method for getting the type of the class

Type* Person_ENTITY::getDirectType()
{
 return (Type*)OC_lookup("Person_ENTITY");
}

//---------------------------------------------
// accessors
//---------------------------------------------
/* ---------------------------------------------------------------------- */

void  Person_ENTITY::SetPersonLevel(int level)
{
 priv_level = level;
}
/* ---------------------------------------------------------------------- */

int Person_ENTITY::GetPersonLevel()
{
 return  priv_level;
}
/* ---------------------------------------------------------------------- */

void Person_ENTITY::SetPersonStatus(int status)
{
  priv_status= status;
}

/* ---------------------------------------------------------------------- */

int Person_ENTITY::GetPersonStatus()
```

```
{
 return priv_status;
}

/* ---------------------------------------------------------------------- */

char* Person_ENTITY :: GetPersonName()
{
 return subordinates_relation_ENTITY :: Name();
}
/* ---------------------------------------------------------------------- */

void Person_ENTITY :: SetPersonName(char* name)
{
 subordinates_relation_ENTITY :: Name(name);
}
/* ---------------------------------------------------------------------- */
```

C. m4 MACEOS:

1. Class Header Macro

```
define(test_header,
`#include <Object.h>
#include <Set.h>

 class $1 : public Object

 {

  private:

   set*      $1_$4 ;
   $2 * $1_$2 ;

  public:

   $1($3, $5);

   $1(APL*);

   Type *getDirectType();
```

```
    void   add_$4($5 * x);

    void   set_$2($3 * x) {$1_$2 = x; }

    $3 * get_$2() {return $1_$4 ;}

    void  remove_$4($5 * x);

    int cardinaality_$4() {return
                        $1_$4->Cardinality();

    OC_Boolean is_transitive_$4_of($5 * x)

    OC_Boolean is_diret_$4_of($5 * x);
      {return $1_$4->Ismember(Entity(x))}
  };')
```

2. Class Definition Macro

```
define(test_body,
`#include <Directory.h>

   $1($3, $5)
    {
     $1_$4= new Set($5);
     $1_$2 = NULL ;
    }

   $1 ::$1(APL *theAPL) : Object(theAPL)
    {
    }


   Type * $1 :: getDirectType()
    {
      return (Type*)OC_lookup("$1");
    }

   void   add_$4($3 x1, $5 * x2)
    {
     if (!$1_$2)
       set_$2(x1);
     $2_$4->Insert(x2);
    }

   void  remove_$4($5 * x)
    {
     $1_$4->Remove(x);
        }')
```

3. Top Level Macro

```
syscmd(echo "test_header(subordinates_relation, team_leader, Person, designer_team,
Person)" | m4 class.h.m4 - > outfile.h)

syscmd(echo "test_body(subordinates_relation, team_leader, Person, designer_team,
Person)" | m4 class.cxx.m4 - > outfile.cxx)
```

C. SAMPLE m4 OUTPUT

1. Sample m4 outpot for the Class Header

```
#include <Object.h>
#include <Set.h>

 class subordinates_relation : public Object

 {

  private:

   set*        subordinates_relation_designer_team ;
   team_leader * subordinates_relation_team_leader ;

  public:

   subordinates_relation(Person, Person);

   subordinates_relation(APL*);

   Type *getDirectType();

   void   add_designer_team(Person * x);

   void   set_team_leader(Person * x) {subordinates_relation_team_leader = x; }

   Person * get_team_leader() {return subordinates_relation_designer_team ;}
```

```
    void   remove_designer_team(Person * x);

    int cardinaality_designer_team() {return
                        subordinates_relation_designer_team->Cardinality();

    OC_Boolean is_transitive_designer_team_of(Person * x)

    OC_Boolean is_diret_designer_team_of(Person * x);
      {return subordinates_relation_designer_team->Ismember(Entity(x))}
  };
```

2. Sample m4 outpot for the Class Definition

```
#include <Directory.h>

   subordinates_relation(Person, Person)
    {
     subordinates_relation_designer_team= new Set(Person);
     subordinates_relation_team_leader = NULL ;
    }

   subordinates_relation ::subordinates_relation(APL *theAPL) : Object(theAPL)
    {
    }

   Type * subordinates_relation :: getDirectType()
    {
      return (Type*)OC_lookup("subordinates_relation");
    }

    void    add_designer_team(Person x1, Person * x2)
    {
     if (!subordinates_relation_team_leader)
        set_team_leader(x1);
     team_leader_designer_team->Insert(x2);
    }

    void   remove_designer_team(Person * x)
     {
     subordinates_relation_designer_team->Remove(x);
     }
```

APPENDIX C. PRIORITY VECTORS COMPUTATION PROGRAMS

A. PRIORITY VECTORS COMPUTATION USING MATLAB DIRECTLY

```
diary resultsM111
finish = 1;
while finish == 1
MainMenu = menu('Data Entry Menu','1- Enter Matrix Elements', '2-Display
Matrix', ' 3-Change Element', '4- Do Eigenvalue Computation','5-
ExitProgram');
if MainMenu==1
  clear A;
  n = input('Input Matrix Dimension  ');
  for i=1:n;
    disp(['ROW Number ',num2str(i)])
    disp('===========')
       for j=1:n;
          element = input(['Enter Element  (',num2str(i),' ,
            ',num2str(j),')']);
          A(i,j) = element;
      end
  end
  disp(A)
elseif MainMenu==2
   disp(A)
   disp('Hit Any Key To Continue');
   pause;
elseif MainMenu==3
   rowN = input('Row# ');
   colN = input('Col# ');
   elementC = input('Enter New Element ');
   A(rowN,colN) = elementC;
 elseif MainMenu==4
     clear X;
     clear L;
     clear lamda_max;
     clear EV;
     clear t;
     clear i;
     clear EVn;
     [X,L]=eig(A);
     max_val = 0;
     for m=1:n
       for k=1:n
         if (L(m,k) > 0) & (L(m,k) > max_val)
           max_val = L(m,k);
           max_col = k;
         end
       end
      end
     lamda_max = max_val
     EV=abs(X(:,max_col));
     t=sum(EV);
     i=1:n;
     EVn(i)=EV(i)/t
  else
   finish = 2;
   disp('Program Terminated Normally')
 end
end
```

B. PRIORITY VECTORS COMPUTATION USING THE CRUDE METHOD OF THE AHP

```
 diary results_defC
finish = 1;
while finish == 1
MainMenu = menu('Data Entry Menu','1- Enter Matrix Elements', '2-Display
Matrix', ' 3-Change Element', '4- Do Eigenvalue Computation','5-
ExitProgram');
if MainMenu==1
  clear A;
  flops(0);
  n = input('Input Matrix Dimension: ');

disp('
    ')
  disp(['Priority Vector Computation Using the Rough Estimates Method of
the AHP for a Matrix of Order ', num2str(n)])

disp('
    ')
  for i=1:n;
    disp(['ROW Number ',num2str(i)])

    disp('============')

    for j=1:n;
       element = input(['Enter Element  (',num2str(i),' ,
',num2str(j),')']);
       A(i,j) = element;
     end
  end
  disp('Comparison Matrix:')
  A
 elseif MainMenu==2
   disp(A)
   disp('Hit Any Key To Continue');
```

```
    pause;
  elseif MainMenu==3
     rowN = input('Row# ');
     colN = input('Col# ');
     elementC = input('Enter New Element ');
     A(rowN,colN) = elementC;
   elseif MainMenu==4
      clear X;
      clear B;
      clear V;
        X = sum(A);
        for m=1:n
         for k=1:n
           B(m,k) = A(m,k)/X(k);
        end
          end
         for  m=1:n
         SumP = 0;
         for k=1:n
         SumP = SumP +  B(m,k);

         end
         V(m) = SumP;
        end
        for k = 1:n
          V(k) = V(k)/n;
        end
       disp('Priority Vector is :')
      V
      V1 = A*V';
      for i=1:n;
        V2(i) = V1(i)/V(i);
      end
      S = sum(V2);
      Lamda_max = S/n;

       disp('Lamda Max Is :')
        disp(Lamda_max)
      disp(['FLOPS COUNT = ',num2str(flops)])
   else
    finish = 2;
    disp('Program Terminated Normally')
  end
end
```

APPENDIX D. PRIORITY VECTORS COMPUTATIONAL RESULTS

```
=======================================================================
```

Example1 for n = 4

```
Input Matrix:

    1.0000    5.0000    6.0000    7.0000
    0.2000    1.0000    4.0000    6.0000
    0.1667    0.2500    1.0000    4.0000
    0.1429    0.1667    0.2500    1.0000

MATLAB Results:

lamda_max =  4.3907
Periority Vector =    0.6187    0.2353    0.1009    0.0451
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
The AHP Crude Method:

lamda_max = 4.4060

Priority Vector = 0.5910    0.2443    0.1151    0.0496

=======================================================================
```

Example2 for n = 4

```
Input Matrix:

    1.0000    0.3333    7.0000    5.0000
    3.0000    1.0000    9.0000    7.0000
    0.1429    0.1111    1.0000    1.0000
    0.2000    0.1429    1.0000    1.0000

MATLAB Results:

lamda_max = 4.0933
Priority Vector = 0.2920    0.5888    0.0553    0.0639
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
The AHP Crude Method:

lamda_max = 4.0939

Priority Vector = 0.2966    0.5802    0.0575    0.0658
=======================================================================
```

Example1 for n = 5

Input Matrix:

```
   1.0000    7.0000    3.0000    7.0000    3.0000
    0.1429    1.0000    0.3333    7.0000    0.1667
    0.3333    3.0000    1.0000    7.0000    0.5000
    0.1429    0.1429    0.1429    1.0000    0.1250
    0.3333    6.0000    2.0000    8.0000    1.0000
```

MATLAB Results:

```
lamda_max = 5.5258


Priority Vector =  0.4580    0.0821    0.1634    0.0296    0.2670
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The AHP Crude Method:

```
lamda_max = 5.5599

Priority Vector = 0.4486    0.0902    0.1676    0.0326    0.2610
=====================================================================
```

Example2 for n = 5

Input Matrix:

```
    1.0000    0.3333    0.1429    0.2000    0.1667
    3.0000    1.0000    0.2500    0.5000    0.5000
    7.0000    4.0000    1.0000    7.0000    5.0000
    5.0000    2.0000    0.1429    1.0000    0.2000
    6.0000    2.0000    0.2000    5.0000    1.0000
```

MATLAB Results:

```
lamda_max = 5.5763


Priority Vector = 0.0363    0.0897    0.5424    0.1057    0.2259

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The AHP Crude Method:

```
lamda_max = 5.6072

Priority Vector = 0.0405    0.0994    0.5124    0.1252    0.2226
=====================================================================
```

Example1 for n = 6

```
Input Matrix:

    1.0000    1.0000    7.0000    5.0000    3.0000    0.3333
    1.0000    1.0000    5.0000    3.0000    1.0000    1.0000
    0.1429    0.2000    1.0000    0.3333    0.1429    0.1111
    0.2000    0.3333    3.0000    1.0000    0.3333    0.3333
    0.3333    1.0000    7.0000    3.0000    1.0000    0.2000
    3.0000    1.0000    9.0000    3.0000    5.0000    1.0000

MATLAB Results:

lamda_max = 6.4750

Priority Vector = 0.2240    0.1915    0.0275    0.0649    0.1325
                  0.3596

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
The AHP Crude Method:

lamda_max = 6.4763

Priority Vector = 0.2233    0.1967    0.0289    0.0686    0.1427
                  0.3399
========================================================================
```

Example2 for n = 6

```
Input Matrix:

     1.0000    0.3333    8.0000    3.0000    3.0000    7.0000
    3.0000    1.0000    9.0000    3.0000    3.0000    9.0000
    0.1250    0.1111    1.0000    0.1667    0.2000    2.0000
    0.3333    0.3333    6.0000    1.0000    0.3333    6.0000
    0.3333    0.3333    5.0000    3.0000    1.0000    6.0000
    0.1429    0.1111    0.5000    0.1667    0.1667    1.0000

MATLAB Results:

lamda_max = 6.4536

Priority Vector = 0.2619    0.3975    0.0334    0.1164    0.1642
                  0.0266
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
The AHP Crude Method:

lamda_max = 6.4616


Priority Vector = 0.2549    0.3889    0.0360    0.1258    0.1668
                  0.0277
========================================================================
```

Example1 for n = 7

Input Matrix:

```
    1.0000    4.0000    9.0000    6.0000    6.0000    5.0000    5.0000
    0.2500    1.0000    7.0000    5.0000    5.0000    3.0000    4.0000
    0.1100    0.1400    1.0000    0.2000    0.2000    0.1400    0.2000
    0.1700    0.2000    5.0000    1.0000    1.0000    0.3300    0.3300
    0.1700    0.2000    5.0000    1.0000    1.0000    0.3300    0.3300
    0.2000    0.3300    7.0000    3.0000    3.0000    1.0000    2.0000
    0.2000    0.2500    5.0000    3.0000    3.0000    0.5000    1.0000
```

MATLAB Results:

```
lamda_max = 7.5996

Priority Vector =  0.4273    0.2304    0.0206    0.0524    0.0524
                   0.1226    0.0943
========================================================================
```

The AHP Crude Method:

```
lamda_max =  7.6062

Priority Vector = 0.4085    0.2264    0.0216    0.0577    0.0577
                  0.1277    0.1002
========================================================================
```

Example1 for n = 8

Input Matrix:

```
    1.0000    3.0000    6.0000    3.0000    7.0000    7.0000    9.0000
    0.3333    1.0000    4.0000    5.0000    5.0000    5.0000    7.0000
    0.1667    0.2500    1.0000    1.0000    0.5000    4.0000    4.0000
    0.3333    0.2000    1.0000    1.0000    2.0000    3.0000    9.0000
    0.1429    0.2000    2.0000    0.5000    1.0000    3.0000    3.0000
    0.1429    0.2000    0.2500    0.3333    0.3333    1.0000    4.0000
    0.1111    0.1429    0.2500    0.1111    0.3333    0.2500    1.0000
    0.1111    0.1111    0.1250    0.1111    0.1429    0.1111    0.1429

  Column 8

    9.0000
    9.0000
    8.0000
    9.0000
    7.0000
    9.0000
    7.0000
    1.0000
```

MATLAB Results:

```
lamda_max = 9.2557

Priority Vector =  0.3617    0.2520    0.0880    0.1186    0.0828
                   0.0529    0.0302    0.0139
======================================================================
```

The AHP Crude Method:

```
lamda_max =  9.3231

Priority Vector = 0.3525    0.2407    0.0905    0.1233    0.0839
                  0.0589    0.034     0.0153
=======================================================================



@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

<u>Example1 for n = 9</u>

Input Matrix:

```
    1.0000    1.0000    1.0000    0.1667    0.1667    0.1667    0.1111
    1.0000    1.0000    1.0000    0.1667    0.1667    0.1667    0.1111
    1.0000    1.0000    1.0000    0.1667    0.1667    0.1667    0.1111
    6.0000    6.0000    6.0000    1.0000    1.0000    1.0000    0.5000
    6.0000    6.0000    6.0000    1.0000    1.0000    1.0000    0.5000
    6.0000    6.0000    6.0000    1.0000    1.0000    1.0000    1.0000
    9.0000    9.0000    9.0000    2.0000    2.0000    1.0000    1.0000
    9.0000    9.0000    9.0000    2.0000    2.0000    1.0000    1.0000
    9.0000    9.0000    9.0000    2.0000    2.0000    1.0000    1.0000

  Columns 8 through 9

    0.1111    0.1111
    0.1111    0.1111
    0.1111    0.1111
    0.5000    0.5000
    0.5000    0.5000
    1.0000    1.0000
    1.0000    1.0000
    1.0000    1.0000
    1.0000    1.0000
```

```
MATLAB Results:

lamda_max = 9.0729

Priority Vector =  0.0206    0.0206    0.0206    0.1129    0.1129
                   0.1442    0.1895    0.1895    0.1895
=======================================================================
The AHP Crude Method:

lamda_max =  9.0731

Priority Vector =  0.0206    0.0206    0.0206    0.1134    0.1134
                   0.1447    0.1889    0.1889    0.1889
========================================================================
```

Example2 for n = 9

```
Input Matrix:

  Columns 1 through 7

    1.0000    4.0000    8.0000    2.0000    4.0000    4.0000    4.0000
    0.2500    1.0000    1.0000    4.0000    0.5000    1.0000    1.0000
    0.1250    1.0000    1.0000    0.1667    0.1667    1.0000    3.0000
    0.5000    0.2500    6.0000    1.0000    1.0000    3.0000    5.0000
    0.2500    2.0000    6.0000    1.0000    1.0000    3.0000    5.0000
    0.2500    1.0000    1.0000    0.3333    0.3333    1.0000    3.0000
    0.2500    1.0000    0.3333    0.2000    0.2000    0.3333    1.0000
    0.2500    1.0000    0.2500    0.1667    0.1667    0.2500    0.5000
    0.1667    1.0000    0.2500    0.1667    0.1667    0.2500    0.5000

  Columns 8 through 9

    4.0000    6.0000
    1.0000    1.0000
    4.0000    4.0000
    6.0000    6.0000
    6.0000    6.0000
    4.0000    4.0000
    2.0000    2.0000
    1.0000    2.0000
    1.0000    1.0000

MATLAB Results:

lamda_max =

Priority Vector = 0.2893    0.1096    0.0695    0.1651    0.1761
                  0.0781    0.0442    0.0366    0.0315

 ======================================================================
```

The AHP Crude Method:

```
lamda_max = 10.8257

Priority Vector = 0.2813    0.0996    0.0745    0.1676    0.1744
                  0.0836    0.0473    0.0391    0.0326
=======================================================================
```

APPENDIX E. COMPUTATIONAL RESULTS FOR THE CASE STUDY

SH1 Judgement Profile

```
Criticism Matrix:

    1.0000    0.3333    2.0000
    2.0000    1.0000    4.0000
    0.3333    0.3333    1.0000


lamda_max = 2.8982

Priority Vector =  0.2609    0.5897    0.1494



Budget Matrix:

    1.0000    0.7500
    1.3300    1.0000


lamda_max =  1.9987

Priority Vector =  0.4289    0.5711

=======================================================================

Safety Matrix:

       1     1
       1     1

lamda_max =  2.0000

Priority Vector = 0.5000    0.5000
========================================================================


Deadline Matrix:

    1.0000    0.6667
    1.5000    1.0000


lamda_max = 2
```

```
Priority Vector = 0.4000    0.6000

SH1 Compsite Priority = 0.47    0.53
=======================================================================
```

SH2 Judgement Profile

```
Criteria Matrix:

    1.0000    2.0000    0.5000
    0.3333    1.0000    1.6000
    0.2500    0.5000    1.0000

lamda_max =  2.4794

Priority Vector =  0.4927    0.3169    0.1904

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


Budget Matrix:

    1.0000    0.3333
    2.0000    1.0000

lamda_max = 1.8165

Priority Vector = 0.2899    0.7101
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Safety Matrix:

    1.0000    1.5000
    0.2000    1.0000

lamda_max = 1.5477

Priority Vector =    0.7325    0.2675
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Deadline Matrix:

    1.0000    0.7000
    1.4000    1.0000

lamda_max = 1.9899

Priority Vector =  0.4142    0.5858

SH2 Compsite Priority = 0.45    0.55
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

+ **SH3 Judgement Profile**

```
Criteria Matrix:

    1.0000    0.5000    0.6667
    3.0000    1.0000    2.0000
    0.5000    0.3333    1.0000

lamda_max =    2.7767

Priority Vector =    0.2325    0.5911    0.1763

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Budget Matrix:

     1     1
     1     1


lamda_max = 2.0000

Priority Vector =    0.5000    0.5000

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Safety Matrix:

     1     1
     1     1

lamda_max =  2.0000

Priority Vector = 0.5000    0.5000

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Deadline Matrix:

    1.0000    1.5000
    0.5000    1.0000

lamda_max =  1.8660

Priority Vector =  0.6340    0.3660

SH3 Compsite Priority = 0.52    0.48

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

SH4 Judgement Profile

```
Criteria Matrix:

    1.0000    1.0000    2.5000
    1.0000    1.0000    1.0000
    0.2000    0.5000    1.0000

lamda_max =    2.6905


Priority Vector =    0.4635    0.3717    0.1648

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Budget Matrix

    1.0000    0.5000
    3.0000    1.0000

lamda_max =    2.2247

Priority Vector =    0.2899    0.7101

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Safety Matrix:

     1     1
     1     1

lamda_max =    2.0000

Priority Vector =    0.5000    0.5000

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Deadline Matrix

    1.0000    0.2500
    3.0000    1.0000

lamda_max =  1.8660

Priority Vector =    0.2240    0.7760

SH4 Compsite Priority = 0.36    0.64

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

SH5 Judgement Profile

```
Criteria Matrix:

    1.0000    0.5000    1.0000
    3.0000    1.0000    2.0000
    1.0000    0.2500    1.0000

lamda_max =    2.9717

Priority Vector =  0.2415    0.5644    0.1941

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Budget Matrix:

    1.0000    0.5000
    2.1000    1.0000

lamda_max =    2.0247

Priority Vector =    0.3279    0.6721

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

safety Matrix:

    1.0000    2.0000
    0.3333    1.0000

lamda_max =    1.8165

Priority Vector =    0.7101    0.2899

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Deadline Matrix:

     1      1
     1      1

lamda_max =  2.0000

Priority Vector =  0.5000    0.5000

SH5 Compsite Priority = 0.58    0.42
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3.	Chairman, Department of Computer Science Code CS Naval Postgraduate School Monterey, California 93943	5
4.	Armament Authority, Training Dept. (Egypt) c/o American Embassy (Cairo, Egypt), Office of Military Cooperation, Box 29 (TNG) FPO, NY 09527-0051	3
5.	Professor Valdis Berzins, Code CS/Bz Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
6.	Professor Luqi, Code CS/Lq Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
7.	Professor Mantak Shinng Code CS/Sh Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
8.	Professor Herschel H. Loomis, Jr, Chairman, Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943	1

9. Professor Qing Wang, Code MR/Wg 1
Department of Meteorology
Naval Postgraduate School
Monterey, California 93943

10. Colonel Osman Mohamed Ibrahim 5
15 Officers Housing, (Apt. 23),
Mazrat El-Haram,
El-Haram, Giza, Egypt

11. LTC. Nabel Khalil 2
SGC# 3079
Naval Postgraduate School
Monterey, California 93943

12. Maj. Hazem Abd El-Hameed 2
SGC# 3078
Naval Postgraduate School
Monterey, California 93943