AFIT/DS/ENG/93-11

ON THE AUTOMATION OF OBJECT-ORIENTED REQUIREMENTS ANALYSIS

DISSERTATION

Nancy L. Crowley, Major, USAF

AFIT/DS/ENG/93-11

AFIT/DS/ENG/93-11

ON THE AUTOMATION OF OBJECT-ORIENTED REQUIREMENTS ANALYSIS

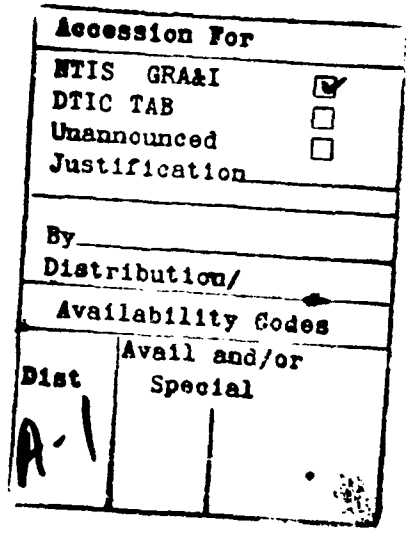
DISSERTATION

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air Education and Training Command
In Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Nancy L. Crowley, B.S., M.S.

Major, USAF

September 1993



AFIT/DS/ENG/93-11

ON THE AUTOMATION OF OBJECT-ORIENTED REQUIREMENTS ANALYSIS

Nancy L. Crowley, B.S., M.S.
Major, USAF

Approved:

	18 Aug 93
	18 Aug 93
	18 Aug 93
	18 Aug 93
	18 AUG 93

Accepted:

13 Sept 93
Dean, School of Engineering

Preface

The correct capture of user requirements is an essential and difficult first step in software development. One method that aids in this process is object-oriented requirements analysis (OORA). This process makes use of method and domain knowledge to develop an object-oriented requirements specification. This research developed an object-oriented model that could be used as a basis for an automated system. An automated system, called the OORA Automated Knowledge System (OAKS), was also developed. OAKS assists in the development of an object-oriented specification through the use of domain knowledge and knowledge of the structure of an object-oriented requirements specifications and the relationships among its components.

There are two people that deserve thanks for their help in getting me through this research. First, many thanks to my advisor, Lt Col Patricia Lawlis, who was always there to lend a hand when I needed it and whose encouragement kept me going. A special thanks to my husband, Bruce, for always being there for me yet giving me the space to work the long hours.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CLOS	Common LISP Object System
OAKS	OORA Automated Knowledge System
OOD	Object-Oriented Design
OODA	Object-Oriented Domain Analysis
OOP	Object-Oriented Programming
OORA	Object-Oriented Requirements Analysis
SA	Structured Analysis

List of Symbols

$\forall$	Universal quantifier, for all
$\exists$	Existential quantifier, there exists
$\rightarrow$	Function
$\Rightarrow$	Implication
$\mathbf{X}$	Cross product
$\in$	Member
$\equiv$	Is defined as
$< >$	Encloses an ordered pair
$\{\}$	Encloses a set
Λ	Binary AND
V	Binary OR
$\cup$	Set union
$\mathcal{N}$	The set of natural numbers
$\Leftrightarrow$	Is equivalent to

Abstract

This research investigated the possibility that an object-oriented requirements analysis (OORA) specification model can be represented in a computer system and used as a basis for the elicitation of the informal information necessary for the development of an object-oriented specification for a particular problem. The proof-of-concept system developed is called the OORA Automated Knowledge System (OAKS). OAKS contains a generic domain model that is modified to satisfy a particular problem in the domain. First, a model was developed that captures the essential characteristics and components of an OORA specification and the relationships among those components. Based on that model, constraints were defined on the order in which the components must be gathered. This information was used as the basis for the elicitation of requirements from the user of OAKS and for the guidelines and rules used in evaluating the object-oriented requirements model. These guidelines and rules were used by OAKS to evaluate the generic domain model and the developing problem model. The OAKS system was then developed and tested based on the model, component order, and the guidelines and rules.

The research showed that such a system is possible and useful. The generic domain model that is used as a basis for OAKS proved to be one that contained all the essential information needed for the development and analysis of object-oriented requirements. The guidelines and rules encoded in OAKS provided the means to maintain consistency and completeness in the OORA model in reference to those rules as changes were made. It was also possible to evaluate the generic domain model prior to its use using these guidelines and rules.

The core of OAKS is a reusable domain model, which represents a domain of interest. The domain model is used as a basis for user changes that are made to meet the specific requirements of a particular problem. The domain model structure was developed to allow it to be ported to other domains of interest and inserted into the OAKS system. Therefore, OAKS represents an OORA system that can be used in numerous domains to develop an OORA specification for a specific problem.

ON THE AUTOMATION OF

OBJECT-ORIENTED REQUIREMENTS ANALYSIS

I. Background and Statement of the Problem

Background

Overview. The object-oriented requirements analysis (OORA) process is one that is crucial to the development of software that meets the needs of the end user. There have been numerous articles and books written on the OORA process and how the process may be aided using a computer-based system. Because of the variance in the different OORA processes there is a need for a well-defined uniform model that can be used as a basis for a computer-based system to aid in conducting an OORA. This research developed an OORA model that was used as the basis for an automated system that assists in the OORA process. The automated system contains information on the domain of interest, the problem to be solved, the OORA process and the resulting specification, and the process of requirements analysis in general.

Object-Oriented Requirements Analysis. OORA is a software development method used to develop a specification of a system's intended behavior. OORA examines the problem domain with the object-oriented perspective of classes and objects [COYO91][BOOC91]. This specification can take any number of forms but represents an object-oriented model of the essential characteristics and behavior of the system. Essential characteristics are those that allow the system to satisfy the user's requirements. OORA is conducted during the requirements phase of the software development process. It is done in place of more traditional software analysis methods, the most popular of which is some

form of structured analysis (SA). Where OORA is based on object abstraction, SA is based on functional abstraction. The two different analysis techniques produce fundamentally different structures and views of the same problem. Although there are a number of methods that have been developed to move from a SA to an object-oriented approach, most researchers agree that the best front-end to object-oriented software development is OORA. [BAIL89] [BERA92a] [BOOC91] [COLB89] [COYO91] [FIRE91] [KURT90]

OORA is used in the analysis phase as a front end to object-oriented design (OOD) and object-oriented programming (OOP). The unifying scheme through the three phases is the use of objects as central elements [KORS90]. The OORA results, however, must be independent of the programming language and the target hardware [BERA92a]. OORA identifies problem space objects, which are entities that the system uses to maintain information or to interface with systems outside itself [JAWO90]. The problem-space objects describe what the system is to do, without addressing how the system is to do it. In contrast, OOD produces solution-space objects. These are derived from the problem-space objects but take into account things such as the computer architecture, performance considerations, and the programming language. The solution-space objects hide the target environment from the problem-space objects and act as controlling objects [WHIT89]. The design process creates an implementation plan for the system by describing how the system is to do its job [KURT90]. After OOD, the system is implemented in a programming language. If that language is object-oriented, this step is called OOP.

In general, object-oriented techniques are characterized by the use of abstract objects with explicit interfaces and the use of inheritance [BERZ89]. There is no one OORA process or even a common set of terminology. There are many different approaches to OORA; these include different steps, models, products, definitions, graphical and textual representations, heuristics, and levels of formality. However, the

processes have a number of basic concepts in common. The following sections discuss the OORA concepts and the steps in the OORA process that will be used in this research.

Object. An object is an abstraction of some entity in the problem domain that encapsulates state and behavior information and has identity. An object is characterized by its state, the services it provides, and the services it requires from other objects to do its function. An object is an abstraction because it represents some real-world entity. The object encapsulates its state and services by controlling access to its state and the services that provide the object's behavior. The state of an object is accessed and modified only by messages that tell the object what to do, but not how to do it [RUBI90]. An object is an instance of a class. Some authors use the terms class and object interchangeably, but especially for OORA, it is important to distinguish between the two.

Class.

> "*A description of one or more objects with a uniform set of attributes and services, including a description of how to create new objects in the class.*" [COYO91]

Classes are the templates for objects. When an object is created, it is an instance of a class, which means it is a copy of the class except it has its own state. The state is the set of values of all the attributes of an object. Classes represent sets of closely related entities in the problem domain. An example of a class would be the class of dogs, with Fido as an instance of that class (an object). Classes are developed in OORA. Objects are not considered until OOD.

Attribute. An attribute is an abstraction of a characteristic that is held by all the instances of a particular class [SHLA89]. Each attribute contains information concerning the set of values the attribute can be assigned, and each object has a particular value for that attribute from that set of legal values. The set of legal values can be any type of value, from numbers and characters to composite types. For example, if an attribute for

the class Dog is Owner, the legal values are the members of the set of names; the object Fido could have the value "John Doe" for the attribute Owner.

State. The state of an object is the collection of values instantiating the set of attributes. If the attributes are represented by {a1,...,an} and the values for the attributes are {v1,...,vn}, then the state = {v1,...,vn}. A class contains the template for the state of an object. The state of an object can only be changed or accessed through the services. If the object Fido had attributes = {Owner, License_Number}, the state could be {"John Doe", 1234}.

Service. Services implement the required behavior of an object [COYO91]. The behavior of an object describes all possible state changes and the method of invocation required for it to do its function. In this research, message passing will be used to invoke services. Services can be represented by functions. A service takes zero or more arguments and returns zero or more results. For example, to change the owner-attribute of the object Fido, the class would have to contain a service that allows the changing of that attribute. A message would be sent to Fido through that service requesting a change of the attribute Owner. For example, to change the value of the Owner attribute, a service is provided called Change_Owner that takes as its one argument the value of the new owner's name. Therefore, to change the value of the attribute Owner to Jill Small, a message would be sent to Fido through the service Change_Owner with the argument "Jill Small". This service may require the use of other services in one or more objects; this would require messages from Fido to those other objects.

Protocol. The protocol of a class is the set of all of its services [RUBI90]. The set of messages the class can respond to maps to the set of services, or protocol, of that class.

Inheritance. Inheritance models an "is a kind of" structure between classes. It is also known as a generalization-specialization structure [COYO91]. Inheritance creates a hierarchy of classes, with the lower classes (subclasses) in the hierarchy inheriting from

higher classes, or superclasses. A subclass is an "is a kind of" or a "specialization of" a superclass. A subclass inherits both attributes and services from its superclasses. A subclass will therefore contain at least the same attributes and services as its superclasses. A subclass may add attributes or services, or it may change the way that an inherited service is implemented. A class can inherit from a single class (single inheritance) or from more than one class (multiple inheritance). An example is the class of Dogs as a subclass inheriting from the class of Animals, the superclass. The class of Animals may have attributes of Number_of_Legs and Natural_Habitat. The class of Dogs may also be a subclass of the class of Pets. This may add the attribute of Owner and add services to access and change owner name. This shows the use of multiple inheritance. The class of Dogs may also add the attribute of License_Number, and it may then add services to access and change the license number. This illustrates specialization.

General Relationships. There are binary relationships between objects that need to be modeled. One such relationship is a "whole-part" relationship where an object is made up of other objects [COYO91]. For example, a plane is made up of wings, engine, etc. Instance connections are relationships in which an object needs an association with another object in order to fulfill its responsibilities [COYO91]. In general, a relationship is an abstraction of an association between two real-world entities that are themselves abstracted as objects [SHLA89]. All relationships are identified by using numeric bounds that show the range of the connections. These bounds are usually finite, although infinite bounds are allowed. For example, a bound may be represented by (1,n), where the lower bound is 1 and the upper bound is some integer represented by n. In general, in a graphical representation a relationship is shown by a line between two objects or classes with a range specified on each side of the connection. For example, assume there exists a relationship R between two classes, A and B. Further assume that the range identified with A is represented by [l:u], where l is the lower bound and u is the upper bound.

Therefore an instance of A has relationship R with as few as l and as many as u instances of B. For example, if A is an object of class Person and B is an object of class Dog, one possible relationship R between A and B is ownership, where A owns B and B is owned by A. The range associated with A might be [0:n], signifying that a person can have as few as no dogs and as many as n dogs. The range associated with B might be [1,1], signifying that a dog can only have one owner and that it must be owned by someone. This relationship is also an instance connection.

Message passing. In the majority of object-oriented techniques, the services of objects communicate with the services of other objects through message passing. When an object's service needs another object's services to complete its processing, it requests those services through a message. The message may pass values, it may return values, it may do both, or it may do neither. The message-connections between objects show processing dependencies and threads of execution. For example, an Animal_Control_Officer object may request the value of the attribute License_Number from the Fido object. There must be a service, such as Get_License_Number, in the class Dogs in order for Fido to respond to this request.

OORA Process Steps. Although there is much disagreement concerning the steps that should be used and their order, there are certain steps that are common to most of the OORA techniques. Most agree that an important early step is the identification of objects and/or classes. This is also commonly accepted as being the most difficult step. The identification of objects requires the use of domain experts that understand the concepts of object-oriented analysis, or OORA experts that understand, in depth, the requirements and peculiarities of the domain [COLB89]. In general, an analyst trained in object-oriented methods will gather all existing information on the domain and the problem to be solved, and extract information from the domain experts through dialog. The analyst will apply all available heuristics and other knowledge of an OORA method to develop an initial set of

classes. There is no standard method for creating a "good" set of classes; neither is there an adequate definition of what "good" is. Different processes are used for class identification. Most use either middle-out, top-down, outside-in, or bottom-up. Middle-out class identification starts with a central, unifying class and works outward by examining classes connected to this middle class. Top-down class identification starts with the system as a class that is broken down into its component classes, and so on, until the lowest level of classes is reached. Outside-in class identification starts with the classes that interface with objects outside the system, then identifies classes connected to those classes, and so on. Bottom-up starts with the classes that do the work of the system, identified through specification and interviews, and connects them. Even though some rules and guidelines have been identified, the OORA processes rely greatly on the experience of their analysts. After the classes have been identified, the static and dynamic class relationships can be identified. The two static relationships are general relationships and inheritance. The two dynamic relationships are message passing and state changes within a class. The relationships are usually described using graphical techniques augmented with textual information in some form. State charts, or some derivation, are widely used to show state transitions. Attributes and services are identified sometime after the classes.

Problems exist with current OORA methods. The problems are the lack of a uniform process or terminology, and the lack of formalization of the process. For example, in most of the current methodologies getting the right information from the domain experts is treated as an art form, not a science. It is heavily based on the experience of the analyst. Knowledge about a problem domain is often implicit and informal [ARAN89]. There is no agreement on what a "good" OORA is or even on what structures, information, and the level of detail it should contain. In most existing OORA methods, terms like "best represent" are used without further definition in selecting

classes. This lack of a good definition of terms is partly due to the immaturity of the OORA process.

Statement of the Problem

The thesis of this research was that OORA process and structure knowledge can be represented in a computer system and used as a basis for the elicitation of the informal information necessary for the development of an object-oriented specification for a problem in a particular domain.

The main objective of the research was to investigate the feasibility of a computer-based system that can guide the conduct of an OORA and result in an object-oriented specification of the system to be developed. The computer based system would use a generic domain model as the template for other problems in the domain. The research developed a basic OORA model that contains the components and relationships necessary for an OORA specification. Using existing OORA processes, guidelines and rules for the evaluation of an existing OORA model were developed. The OORA model and guidelines and rules were then used as a basis for the computer-based system.

II. Literature Review

Overview

This research used technology from a number of different areas. These are:

- Requirements analysis
- Requirements analysis systems using artificial intelligence technology
- Object-oriented domain analysis
- Object-oriented requirements analysis
- Existing systems that use a combination of software engineering and artificial intelligence techniques to conduct a requirements analysis

Each of the above technology areas are discussed in a separate section. The first section discusses the general process of requirements analysis and the general nature of software specifications. Next discussed is the general nature of computer systems that conduct requirements analysis and how knowledge-based techniques can be used in the process of requirements acquisition. In the next two sections the general processes of object-oriented domain analysis and object-oriented requirements analysis are discussed. These two processes form the basis for the knowledge in OAKS. The fifth section summarizes the research on existing systems that combine software engineering and artificial intelligence techniques for conducting a requirements analysis. After all five technology areas have been discussed, the last section in this chapter discusses the relationship of the technology areas to OAKS.

Requirements Analysis

Requirements analysis is the process of eliciting the user's needs, determining what information goes into the software system, and representing that information in a requirements specification. The requirements elicitation process is the first part of requirements analysis. It is the process that elicits the information from the user on what to build [HOLB90]. After the elicitation phase, the analyst must determine what information is required to be in the requirements specification. The requirements specification is a precise set of statements about the intended behavior of the system [ZERO91].

The process of requirements analysis is considered to be that of building a conceptual model of the system to be developed. The conceptual model is a model of the application domain as perceived by both the users and the analysts. It is the common understanding that the users and the analysts have of the system to be developed and is therefore the basis for communication between the analysts and the users. The conceptual model is used as the basis for all further development, including design and coding [KUNG89].

The requirements analysis phase of the software development process is crucial because the effective transfer of knowledge from the problem domain to the requirements specification is a prime factor affecting the quality of the software system [CARV90] [ZERO91]. Studies indicate that errors in requirements are more costly than any other kind of errors [REUB91]. Therefore, there will be greater gains in productivity and quality in assisting the requirements analysis process than in assisting the design and coding processes, because the requirements document is the basis for all later processes [LOUC88].

Informality will always exist during the requirements analysis process and is inherent in the initial stages of the process. This is because the process is primarily cognitive and deals with material which is uncertain, unreliable, and inconsistent [KUNG89] [ZERO91].

The question is whether the informal form will only exist outside a computer system in someone's head or in informal, unanalyzable documentation, or whether this informal form can be explicitly entered into the computer and transformed into a more formal form [BALZ78].

Informality is portrayed in the initial stages of the process, as the user makes postulations about the functionality and constraints of the system to the analyst. This knowledge is often incomplete, fuzzy, and incoherent. This is not a fault of the process, but the nature of how humans deal with complexity. Informality is an essential part of the human thought process. The process is to start with an almost-right description and then incrementally modify it until it accurately represents the system to be developed [REUB91]. Therefore, the requirements analysis process will require a series of feedback analyses and refinements of the user's initial concept of the system to a more complete, unambiguous form [BOBB90]. This transformation process can be made more efficient through the use of a computer-based aid that will accept the informal form and aid in the transformation to the formal. This requires the development of formalized methods used to gather the informal form. Examples of these methods are structured question and answer sessions and formalized languages.

The process of converting from the informal requirements gathered from the domain experts to a requirements specification is error-prone and labor-intensive. The activities involved in this process are knowledge-intensive, informal, human-intensive, and largely undocumented. There is a need for a system that manages these knowledge-intensive activities [BALZ85]. The system must provide mechanisms which encourage the development of informal models and assist in experimentation prior to developing a formal specification [LOUC90].

The requirements analysis process is a problem-solving process that derives a statement of the problem, which is refined into the conceptual model of what is needed.

This requires the real world to be mapped into some requirements specification language or representation and that representation communicated to the user. The user then validates the conceptual model to ensure that it meets the user's requirements. The purpose of the process is to evolve the initial requirements statements into a state of consistency and adequacy [LOUC88].

There are three basic phases in the requirements analysis process:

(1) Elicitation. The analyst acts as a facilitator gathering the requirements information from the user, with the product an informal specification.

(2) Formalization. The analyst creates a formal or unambiguous specification from an informal one.

(3) Validation. The confidence that the specification conforms to the user's desires is increased in this phase [REUB91].

The process of writing a requirements specification requires several iterations of these phases to get the content of the specification to match the user's intent [BALZ85]. The process requires knowledge on how to interact with a user to extract the initial set of requirements, how to take those requirements and build a specification, how to recognize an incomplete, inconsistent, and unresponsive specification, and how to explain the specification to the user and validate it. The process also requires a thorough understanding of the application domain.

The hardest part of the requirements analysis process is the elicitation of the requirements from the users. Current techniques require a knowledgeable analyst that is able to extract the proper information from the users. The users often only have a vague notion of what they want and a narrow view of what is possible. They bring with them the expertise in the domain where the analyst has expertise in the requirements analysis process. Therefore the analyst must have expertise in pulling out the necessary information and filling in details that are found to be missing [FICK88].

There are some basic characteristics that expert analysts share:

(1) They use hypothetical examples to explain concepts to the users, to argue for or against inclusion of a component, and to refine their understanding of a problem.

(2) They are aware of higher-level policy issues in a domain and are able to use this knowledge to include or exclude components.

(3) They use summarization to verify their understanding and to verify that they have covered all concepts [FICK88].

(4) They use analogy to relate current problems to previous experience.

(5) They construct hierarchies of concepts starting with an abstract mental model and refining that into a concrete model.

(6) They use domain knowledge. They construct complex artifacts by using their previous experience. It is difficult to acquire new knowledge unless one already has a large amount of relevant old knowledge.

(7) Their reasoning process is guided by some underlying generic process appropriate to the task and the domain [LOUC90].

The analyst uses informal communication with the users to gather information. During this informal communication, the users employ language containing special words and jargon, ambiguity which must be disambiguated by using the surrounding context, statements in poor ordering, and contradictory information. The information provided by the user is often incomplete and can be inaccurate in that it does not reflect what the speaker had in mind [REUB91]. These problems must be addressed by the analyst.

The specification that is a result of the analysis process and is used as input to the design and coding phase can take a number of forms. Specifications are, in reality, programs written in a very high level abstract programming language [BALZ78]. This language can be in natural language form, in the form of a formal specification language, or defined using formalized representations. These formalized representations include

petri nets, frames, rule-bases, regular expressions, transition diagrams, state diagrams, data flow diagrams, etc. Each of these representations has different properties that allow different degrees of formality and different theorem-proving properties. The specification should capture the observable behavior of a system and allow all valid implementations.

In addition to its final form, the specification may have other forms during its development. For example, if the final specification form is some formal specification language, the specification may start in some informal form and then be converted into the formal specification language.

A specification form must have certain characteristics:

(1) It must be able to express various aspects of the specification, and then have the ability to combine them.

(2) It must be able to be used in different and varying domains.

(3) It must not force a certain sequence of decisions that may force the users to make decisions they don't wish to make or are not entitled to make [BABB85].

(4) It must be testable and modifiable.

(5) It must specify what the system is to do, and not how it is to do it.

(6) It should be a cognitive model, not a design or implementation model.

(7) It should be tolerant of incompleteness and augmentable. No specification is ever totally complete because of the complex environments it models [BALZ79].

As stated, the specification can be represented by the use of a formal specification language. The use of formal specification languages require the entire specification to be conceived at once. A better method is through gradual elaboration, where the process starts with a simplified kernel, and then expands out. This requires an incremental specification language such as Gist [BALZ85]. Also, formal specification languages are not good for communication with the user and do not provide mechanisms for decomposing the real-world problem. As a result, the use of formal specification

languages is usually preceded by the application of an informal method [CARV90]. There are three basic sets of tools that can be used to check a formal specification:

(1) Theorem provers that prove that all behaviors have some desired set of properties. The problem is that it is hard to characterize the behavior using the properties, and theorem provers only check the expected.

(2) Interpreters for specification languages that allow the languages to be executable. These provide narrow case-by-case feedback on the testing of the specification.

(3) Symbolic evaluation that allows the test cases for the specification to be partially specified. Those aspects not specified are treated symbolically. Therefore, entire classes of test cases can be explored automatically.

Instead of using a formal specification language, natural language can be used as the form of the specification. The main difference between a natural-language specification and the formal equivalent is that partial descriptions rather than complete descriptions are used. The partial descriptions can be completed from the surrounding text by a computer system. The completion of the partial descriptions may produce zero, one, or several valid interpretations. The partial descriptions focus attention on the relevant issues and condense the size of the specification. Formal specifications do not have these properties [BALZ78].

A desirable characteristic of a specification is that it be operational. If the specification is operational, it can be used to prove that a proposed implementation satisfies the specification. This means that it must be able to generate possible behaviors among which must be the proposed solution [BALZ79]. If the specification is operational, it can be directly evaluated as a software prototype. An operational specification can be translated into code that will preserve program correctness. [TSAI89]. The problem with operational specifications is that they are hard to construct because they are formal. Every reference to an object or action must be consistent and complete [BALZ78].

Another desirable characteristic of a specification is that it be executable. This requires it to be represented by some executable language. To make a specification language executable, it can either be a wide-spectrum language or an interpreted language. A wide spectrum language contains both low-level and high-level constructs. The low-level constructs in the language can be directly executed. Interpreted languages are declarative, but their constructs can be given operational interpretations. An example of an interpreted language is Prolog, which is based on Horn clauses. Requirements expressed in Horn-clause logic are executable using an abstract interpreter (as in a Prolog program). Using the Prolog theorem-proving mechanism, the validity of the requirements can be checked. The requirements can also be checked against the domain model by formulating goal clauses expressing some fact about the system and determining whether that fact can be derived from the requirements. The problem with Horn clause logic is that there is neither an inheritance mechanism nor an exception mechanism to its rules and constraints [TSAI91]. The use of Prolog therefore requires the coding of structures to add these capabilities.

Requirements Analysis Systems Using Artificial Intelligence Technology

The greatest source of variance in software productivity is in the differences in skill among analysts. Applying knowledge is the key to developing effective software. Therefore, developing software is a knowledge-intensive activity [LOUC88].

A system that captures the entire requirements acquisition process, starting with the informal requirements, would be very useful because of the criticality and the complexity of this phase of the software development process. Knowledge representation techniques can form a useful platform for such a system because of the nature of the software analysis process [IPCH91]. Requirements analysis is a cognitive activity that is based on informal

models of domain knowledge, problem-solving knowledge, analysis process knowledge, and general software problem-solving knowledge.

In the process of developing a software specification, the analyst uses knowledge on how to conduct a requirements analysis with associated heuristics. This knowledge is suited to being represented in a knowledge-based system. A knowledge-based approach can also provide useful tools in the process of capturing requirements from a user. This approach offers the facilities needed for the acquisition and validation of requirements, such as rapid prototyping, knowledge bases, intelligent interfaces, and heuristic approaches. Paradigms like restricted Horn clauses, semantic networks, and rule-based production systems are suitable for declarative representation of knowledge. Frames combine declarative, inheritance structure and procedural knowledge [BOBB90].

There are a number of categories of knowledge that could be embodied in a requirements analysis system. The categories are: the domain, the environment in which the software is to operate, the software requirements analysis process itself, knowledge elicitation techniques, informal and formal models of the specification, and transformation techniques among the various representations.

If the system starts with the user supplying the formal model, there is no requirement for knowledge elicitation techniques. Otherwise, the system has to have knowledge on how to elicit the information it needs in order to develop a requirements specification. Artificial intelligence techniques in knowledge acquisition, natural language interaction, and question-answer systems can be used.

How much (if any) domain specific knowledge needs to be in such a system is arguable. It may be argued that domain-specific information could be entered into the system as part of the specification. In this scenario, the user would enter both domain and problem information and would therefore have to be a domain, problem, and requirements process expert. The problem is that domain-specific knowledge consists not just of

definitions, but also of general knowledge of prior typical systems. It is also difficult for the computationally naive user to express domain knowledge unless the system already knows a significant amount [BARZ85] [CHIN89]. The system would have to contain extensive domain knowledge in order to operate with the computationally naive user and develop a specification.

Another scenario is that the user understands the domain but may not possess the knowledge to explain in proper form the task the system is to perform and the method to solve the problem. The system would emulate an expert analyst that has limited domain knowledge but has knowledge about the requirements analysis process and how to elicit information from the user. In this case the system would contain knowledge concerning the form that the requirements specification is expected to take, the process by which the specification evolves, and the process of eliciting the information from the user. The system would not contain much domain knowledge. Therefore, the system would judge the quality of the specification without necessarily understanding its contents [SCHO91].

In gathering the required information from the user, the system might assume a number of different roles. Its role could be passive, in that it provides tools to enter the constructs but does not guide the process in any way. In such a role, it might do checking on the completed specification or portions of the specification, and may transform the entered specification. The user of such a system would have to have both domain knowledge and process knowledge on how to build a requirements specification. Systems that take on a passive role are those that take as input natural-language text and those that provide a formal specification language for building a specification.

A second role could be that used in "sloppy" modeling, where knowledge acquisition is viewed as a cooperative process between the user and the system. The user is not required to develop a complete and well-structured model before interacting with the system. The emphasis is on a cooperative, mixed-initiative modeling process. There is no

fixed, unchangeable dialog, so the user can use any of the system's facilities at any time. The system tries to organize and complete the knowledge entered by the user. The system offers small operations, used to create the specification components, that can be done at any time and in any order. The information that is entered into the system can be changed at any time. The system tries to maintain integrity and consistency of the information and provides immediate feedback on the consequences of all operations. The system allows the user to view the evolving model at any time, and supports different user levels from user-controlled modeling to mixed initiative modeling. For this type of system the user would have to have domain knowledge and at least some process knowledge [WROB88]. The "sloppy" modeling approach is the basic approach used in OAKS.

A third approach is to have the system guide the user through the steps of gathering the knowledge necessary to create the specification. The system does not enforce a rigid sequence of steps; it allows the user to go back to add and modify previously entered information. However, it is more rigid than sloppy modeling systems in that the system is the guide for the acquisition of the information and contains the knowledge on how a model should be built. The assumption underlying such a system is that the user has domain knowledge but not process knowledge.

Object-Oriented Domain Analysis (OODA)

OODA is the process of identifying the objects, operations, and relationships in a problem domain so they can be reused in software specification and construction. A problem domain is based on a shared understanding in a community that includes a shared vocabulary, shared semantics, and a shared knowledge of domain concepts and methods [ARAN89]. Therefore, there are identifiable experts in a domain, and a set of related problems to be solved in the domain.

OODA is distinguished from OORA because OODA is separate from any one problem. It seeks to find a set of objects, operations, and relationships that are common across a number of problems in an application domain. The results of OODA can be used in OORA, and the main purpose of OODA is the reuse of the common structures in many OORAs for various problems in the domain. Therefore, most of the discussion on OODA in the literature occurs in the context of reuse.

As with OORA, there are different approaches to OODA and different terminology used. Some authors [SHLA89] use domain analysis as a synonym for OORA because OORA requires domain analysis in the problem domain in order to identify the structures of OORA. The difference is that in OORA, a specific problem is being addressed, therefore objects and relationships in the domain but not needed by the specific problem are not addressed at all or are addressed in lesser detail. OODA examines all possible structures in a domain and tries to determine which should be modeled for later reuse in OORA.

The result of an OODA is a general object-oriented model of the problem domain. OODA can aid in OORA by the identification of complete, robust objects and their interactions [BERA92b].

Object-Oriented Requirements Analysis (OORA)

Background. Requirements analysis is the study of a problem domain leading to the specification of observable behavior. It is the process of extracting the needs that the system must fulfill. [COYO91] OORA conducts requirements analysis with the object-oriented perspective of classes and objects and their relationships. In OORA, the classes and objects that best represent the problem are developed [WALT78].

Any model of the OORA process should be unambiguous, abstract and consistent [HAYE91]. It should be unambiguous in that there is only one meaning for everything in the model. It should be abstract in that it represents real-world entities that are needed to satisfy the requirements and it does not contain any implementation information. The model should be consistent so there are no conflicting requirements.

Many models have been proposed for the OORA process. They accomplish different steps in different orders. There is, however, some commonality among many of the concepts in the different models that are used in conducting an OORA [BERA92a] [BOOC91] [BULM91] [COLE92] [COLB89] [COYO91] [HAYE91] [JAOL89] [LADE89] [SHLA88] [YAUL88]. These concepts are:

- Finding classes and objects.
- Defining the inheritance structure.
- Defining object relationships.
- Defining class attributes.
- Defining class services.
- Developing a state model of each class.

In each of the models, these concepts are not necessarily examined in the order shown, or sequentially. They are examined iteratively, with each concept having influence on the others.

Not all of the concepts discussed by the different models described in the literature are listed above. In some of the models, the concepts are grouped together into one step or broken into more than one step. The concepts, or possible OORA steps, listed here represent the most common and basic concepts in the various methods. Not used as references in the development of these basic concepts were those articles that described methods that use the results of a structured analysis as a basis for an OORA or OOD. The methods referred to here start with an informal requirements document developed by the

user, with supplemental information from general domain documentation, prior projects in the domain, and direct discussions with domain experts.

Besides the differences in the concepts, the terminology used by the various referenced methods was not consistent. Therefore, the terminology defined in Chapter 1 will be used.

Finding Classes and Objects. The classes are considered during analysis and object instances during design, although when examining a problem domain to uncover classes, sets of objects are looked for. To avoid confusion, the terminology of [COYO91] will be used. The term "class-&-object" will be used for a class and its instances. Class-&-objects are structures, systems outside the system under consideration, devices that the system needs to interact with, a time that needs to be recorded, an event that needs to be recorded, human roles, organizational roles, operational procedures, physical locations, specifications, quality criteria, aggregations of equipment, steps in a process, tangible things, interactions between two or more objects, or something toward which thought or action is directed.

One method for finding classes and objects is to start by identifying class-&-objects that are associated with interfaces between the system and the outside world, or with messages the system receives from its environment, and then to work in by identifying those class-&-objects associated with the classes already identified [BULM91] [MRDA90] [ROSS90]. Another method is to start with the top level class-&-object, which is the overall system, and then break out the class-&-objects hierarchically, by having class-&-objects at a higher level composed of class-&-objects at a lower level [COLB89] [ROSS90]. Another method is to build models of the problem and derive the class-&-objects from the models [COLE92]. Still another method is to identify class-&-objects directly from the problem information by looking for class-&-objects in the various categories [COYO91] [JALO89]. A final method is the identification of nouns in the problem information as potential class-&-objects [BOOC91] [WHIT89].

Once the initial set of class-&-objects is identified, there are various processes used to refine the set by adding, deleting, and combining class-&-objects. Objects are also categorized in some of the methods.

Class-&-objects can be defined as active or passive. An active class-&-object is one that can act without any outside stimulus or initiation. A passive class-&-object only acts when motivated by an active object [COLB89]. Another way to classify class-&-objects is as directors, servers, or agents. A director is an active object that sends messages to other class-&-objects based on external or internal stimulus. Agents are like passive objects in that they do not initiate action until called. Servers are passive objects that do not send messages to other class-&-objects [WALT78].

New classes can be created by combining classes, creating a subclass of an existing class, or breaking a complex class into a number of smaller, cooperating classes. [RUBI90] A class-&-object should have at least two instances, otherwise it should be combined into another class-&-object [WIRF90].

There are numerous other heuristics for deciding whether a class-&-object is "appropriate" for the problem, e.g., that a class-&-objects should have more than one attribute.

Defining the Inheritance Structure. Inheritance structure is an "is a kind of" relationship between classes. The inheritance structure may be between classes already identified, or between classes identified and other classes yet to be uncovered.

One method for identifying the inheritance structure is to start with the existing classes and see how they are related to each other, determine if there are potential generalizations of the class that may be a superclasses, and examine the possibility that there are specializations needed of the class. Any new class will have to be examined to see if it meets all the requirements of a class and that it is in the problem domain [COYO91] .

Another related method is to find similarities and differences between the uncovered classes to build a class inheritance hierarchy [LADD90].

The class structure is normally represented as a graph, with the superclasses on top and the subclasses underneath. There is a line between each class and the class or classes from which it inherits. Figure 1 shows an example class structure using multiple inheritance. Class A is the direct superclass of Classes B, C, and D. Class E inherits from both Class B and Class C (multiple inheritance). In actuality, Class A is a superclass for all of the classes in the Figure because all the classes inherit the attributes and services of Class A.

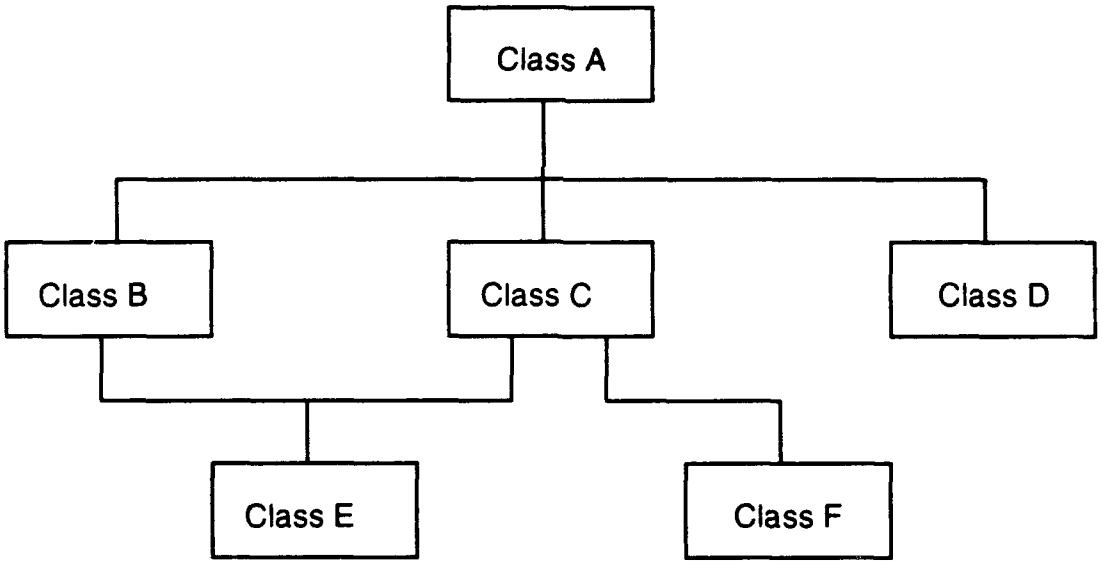


Figure 2-1. Inheritance Structure

There is no one perfect inheritance structure for a problem. There are good and bad structures, depending on how well the problem domain is modeled and the results of the

evaluation of the structure using other metrics such as the coupling between classes. Classes should be as independent and self-contained as possible. A class should need information through message passing from as few other classes as possible and should not depend on any other class's internal structure. There are tradeoffs on the depth of the inheritance structure. Class structures that are wide and shallow have classes that are fairly independent and therefore are not as likely to require changes when other classes change. These classes can also be used in different ways without having to rewrite the class. Class structures that are deep exploit the commonality between classes, so each class contains less information than classes in the wide and shallow structure, but the classes are very dependent on each other [BOOC91].

Defining Object Relationships. Class-&-objects can have different relationships, other than that of inheritance. An object relationship is an abstraction of an association between two real-world entities that are abstracted as class-&-objects. Relationships are labeled with numeric bounds that show the range of the connections.

Whole-part relationships can be identified by looking for assembly parts, container contents, and the members of collections. If those entities that make up the whole-part structure are candidate class-&-objects, and they are part of the problem domain, then they should be part of the model (if they are not already) [COYO91].

If two class-&-objects are related, a new class-&-object should be added that contains the relationship information. This makes the classes independent and therefore more usable [BULM91]. For example, a relationship between the class-&-object Cars and the class-&-object People is ownership. A person can own zero or more cars, and a car must be owned by one person but can be owned by more than one person. This may be modeled by:

ownership

People ---------- Cars

[0:n] [1,n]

The OORA methodology adds a new class-&-object that contains the information on who owns what car. This may be called "Car_Ownership" and will contain all information about car ownership, instead of the information being spread across two class-&-objects. This new class-&-object is called an associative class-&-object [SHLA89].

These relationships can be found in the documentation or by asking domain experts the possible relationships that already identified class-&-objects have with other entities.

Relationships in OORA are similar to the entity-relation diagrams used in structured analysis methods. In entity-relation diagrams, entities, which are real world objects, are shown as rectangles and the relationships between the entities are represented as diamonds [DAVI90]. The entities would correspond to classes in OORA and the relationships to object relationships.

Defining Class Attributes. Attributes define how a class-&-object is viewed in the domain by defining the characteristics of the class-&-object that are important in the problem. For example, there are many possible attributes for the class-&-object Person, such as name, address, phone, height, weight, marital status, hair color, place of birth, etc. The key is to pick those attributes that are required in the problem. For example, if the problem is a mailing system for school announcements, probably only name and address would be required. It is highly unlikely that the system would have to keep information on hair color, height and weight.

The attributes are defined by the analyst examining what the objects of a class are responsible for knowing or keeping information about over time. The analyst determines what subset of the object's attributes is needed in the problem domain. Since the values of

the attributes make up the object's state, possible states the object can be in during its lifetime must be examined. A key is to look at what the system needs to know about that object [COYO91].

Each attribute should be a single value or a tightly related group of values. The actual way the attributes will be identified and stored will not be determined until OOD [COYO91].

A set of attributes should completely describe the necessary state, each attribute should capture a separate concept, and attributes should be independent of each other. Attributes can be placed in three categories. Descriptive attributes are those that can be used in the sentence, "The ATTRIBUTE of OBJECT is .....". For example, "Color" is an attribute of the class-&-object "Dress", since it can be used in the sentence, "The color of the dress is red." Naming attributes are arbitrary names and labels, such as a social security number. Referential attributes are facts that tie one object to another. An example is the class-&-object Student, which may have the attribute School_Name that would tie that student to a particular school. The domain of each attribute, or the set of values each attribute can take on, must be able to be identified. Attributes must represent a characteristic of the entire class-&-object, and not just of another attribute. For example, if there is a class-&-object Student with the attributes Name and School_Name, there should not be an attribute of School_Address. This attribute only applies to the attribute School_Name, and not to the entire class-&-object. School_Address should be an attribute of a class-&-object School which also includes the attribute School_Name [SHLA88].

Where the attributes are placed is determined by the class-&-object to which the attribute is most tightly related. The attribute should be put on the highest level of the inheritance structure where the attribute applies. An attribute should always have a value in an object. If there is an attribute that does not apply to some of the objects, either the

attribute is in the wrong class-&-object or the inheritance structure needs to be modified [COYO91].

Defining Class Services. The services contain the necessary processing for a class-&-object. Any change of state is accomplished through the execution of a service. The information required about a service are the service's name, input (if any) and output (if any), services needed from other objects, and the state change caused by the service (if any). The specification of how the service is implemented is left to OOD and OOP.

Before the services can be defined, the possible states an object can go through must be defined. For example, if Address is an attribute of the class-&-object Person, an address change of a particular Person object would cause a change of state. This address change can only be done through the invocation of a service. State models are used to show state transitions for an object.

Once the state transitions have been defined, the services can be identified. Services can be categorized into algorithmically-simple services and algorithmically-complex services. The algorithmically-simple services are those that create a new object in the class, connect or disconnect an object with another, access or change the attribute values of an object, or release or delete an object. An example of an algorithmically-simple service is one that changes the address of a Person object, given that Address is an attribute. The algorithmically-complex services calculate results from attribute values or monitor an external system or device. An example would be a service that calculates the pay based on the hours worked, the dollars per hour, and the tax tables. Services are identified by looking at the categories and the required state transitions and determining what is required for each class-&-object [COYO91].

Related to the identification of services is the identification of the required message connections. If a service requires information from another object, it sends a message that invokes that object's services. The message connections show the processing

dependencies between objects. For example, if a Paycheck object needs to calculate an employee's pay, it may have to access the Employee object, and a State_Tax table object, and a Federal_Tax table object to get all the information required. It may be desirable to show the timing relationships between objects through the message connections. For example, if a User object requests information through the use of a service, the trace of message flow through the system could be analyzed by tracing the messages starting with the messages invoked by that service through all messages required to fulfill the requirements of the first service call.

The services should provide all the processing needed for an object, but there is disagreement on the level of the services. In [COYO91], a basic set of services is emphasized, whereas [BULM91] states that the higher the level of the operations in the object, the better. For example, if the object is a stack, two services could be Get_Top and Pop. These two could be required to pop the stack and get back the item at the top of the stack if basic services were provided. On the other hand, one service could be provided that does both functions, if that is all the object will be required to do and higher-level services were used.

One method recommends that if an object has too many operations, it should be examined for decomposition into several smaller objects. Also, an object should not contain both high-level and low-level operations. Two class-&-objects should be created: one with the high-level operations and one with the low-level operations [WHIT89].

The number of message connections that a class-&-object has with other class-&-objects should be minimized. This minimizes the system coupling and makes for a system that is easier to test and change [CHID91] [WIRF90].

Developing a State Model For Each Class. The state of an object is the set of values of its attributes. The state can only be changed through invocation of a service. Most models provide a notation for describing the possible state changes that an object of a

class can go through. This is also called the dynamic behavior of a class [BERA92a] [COLB89] [HAYE91] [SHLA89].

There are often restrictions on when an object can change state, or, in other words, when a service can change the state. These restrictions are typically portrayed as either preconditions on the service or annotations on a graph showing possible state changes. Pre- and post-conditions can be written for each service. These declare when state changes are allowed (precondition) and what changes are made it the precondition is met (postcondition). Given some initial state, the dynamic behavior can be determined using these preconditions and postconditions. The dynamic behavior can also be shown graphically using state transition diagrams, statecharts, or variations. These show all possible states of a class and all possible transitions between states. The transitions are labeled with the event that causes the transition and the conditions for the transition to occur.

Some OORA models do not address the dynamic behavior to the degree of showing all state changes. Coad and Yourdan's approach shows the message passing possible and the services, but not all possible states or preconditions and postconditions.

Existing Systems

Existing systems that support parts or all of the requirements analysis process are briefly described. They are placed into one of three groupings. Members of the first group use a natural language front end, those of the second use a constrained input, and those of the third use formal specifications as input.

Systems That Use a Natural Language Front End.

[ARIN89] presents a natural language front end for knowledge acquisition for a knowledge base. Because the system uses natural language, it is constrained to a limited

problem domain. Natural language, being characterized by ambiguity, idiomatic expressions, and context dependence, requires a restricted domain for the systems to be able to disambiguate the input text with low failure rates. The system maps from natural language to a knowledge representation in the knowledge base. The system includes expertise in the knowledge elicitation task. The system does consistency and integrity testing, which is where much of the difficulty is. The system is intended to greatly reduce the role of the knowledge engineer in the initial acquisition and totally replace the knowledge engineer after the system has been implemented and initially tested. The system must be pre-calibrated with high-quality basic domain knowledge for the natural language front end to be able to accurately parse the user input. The knowledge is represented in Prolog.

The SAFE system takes a natural language text that has been parenthesized to show the sentence structure as input and produces a formal operational specification (it has executable semantics). The natural language input must be small (about 10 or so sentences). The parentheses are used to avoid syntactic parsing problems. The system resolves issues such as missing operands, incomplete references, and terminology changes. The operational specification is in a language called GIST [BALZ78] [BALZ85].

MOANA uses natural language dialog to acquire formal software requirements. In order to understand the user, MOANA uses knowledge about the structure and requirements of typical software systems. MOANA is not domain specific. It uses natural language without constraining the domain because it does not have to completely understand the entire natural language input. The user of MOANA is a domain expert who is not a software engineer. The dialog is designed to avoid unconstrained textual input by controlling the initiative in the dialog. The system looks for keywords that can be matched to a set of stereotypical software models. These models are used as a starting point for building models of the user's desired software system and consist of necessary,

typical, and optional components. MOANA has a script that specifies the type and order of information that is obtained from the user. The system asks clarifying questions and identifies incompleteness and inconsistencies. It uses a natural language generator to communicate with the user. The system requirements are represented using operational and data flow models. The output of MOANA is a series of software models which is fed to the software designer [CHIN89].

IDeA is an environment for supporting high-level specification and design. It provides graphical support for data flow diagrams and a natural language front end for interpreting informal specifications. Its unifying model is based on data flow representations and methodologies. There are generic data flow diagrams in the system that are instantiated to create the design [LUBA86].

The unnamed system in [DOHE90] takes as input a functional specification written in English. It reduces ambiguities and shows the revised sentences to the user for review. Once they are accepted, the sentences are converted into predicate form. Usually the initial information entered is incomplete and inconsistent, so the system uses internal domain knowledge or queries the user. The domain knowledge in the system is represented in Prolog structures in a conventional tree inheritance structure. The goal is to get the information to a point where an established design algorithm can be used.

[SAEK89] discusses parsing a natural language specification with human interaction to develop an object-oriented specification that can be transformed into a design specification, and then into code. The system concentrates on extracting verb phrases. The user has to decide which of the extracted nouns and verbs are important, and therefore the system relies heavily on user interaction. There is domain information in the system and no class hierarchies are produced. The system is used for time-oriented systems.

KAPS is a system that provides knowledge-based assistance to the requirements phase from an object-oriented perspective. The goal is to add formalism - not to produce a formal specification. The system accepts the user's natural language description of a system's behavior. The input must be grammatically correct with no pronouns. The system does an interactive parse and produces a standard Lisp expression. Using the parsed sentences, facts are asserted into the knowledge base. The result is object-oriented, although the model is not a complete object-oriented model. Feedback with the user refines the model. Knowledge about OORA is not encoded. No domain knowledge is contained in the system [CARV90].

Systems That Use Constrained Input, Not Formal Language, as the Input Form.

[BARS85] discusses an automatic programming system that starts with an informal specification consisting of preconditions, postconditions, inputs and outputs that are in a higher order language (HOL)-like form. This form is transformed into a formal specification and then coded. This is done using knowledge about programming, the application domain, manipulating mathematical expressions, and the target architecture and language. The informal input is formalized by either recognizing the informal form and replacing it with a formal one, or trying to apply problem-solving heuristics to decompose the problem into smaller ones. The domain-independent problem solving heuristics are represented as pattern (the informal input) - action (decomposition or a formal specification) rules. The basic facts and relationships of domain knowledge are represented as structured objects and stored in a knowledge base. The system is dependent on being able to operate in a very narrow domain.

In [BOBB90], the software requirements are elicited from the end user using a logic-based, declarative tool. The attributive information (objects and attributes) is elicited and represented using Prolog. This produces Prolog programs that can be analyzed for consistency, completeness, omission, and ambiguity through execution of the

representation. The data elicited is in rule and fact form. The behavior is elicited in frame form.

ORM (object relationship model) is an alternative model to object-oriented modeling that uses complex, or composite objects. It uses the standard object-oriented concepts of objects, relationships, and classes. A role is assigned to each class in a relationship and cardinality constraints are specified for both classes involved in the role. Attributes are considered to be a special kind of object class. The objects and relationships are grouped to form complex objects. The analyst enters the information in graphical form and then asks for an evaluation of the model. A heuristic approach is taken in evaluating consistency and completeness, using forward chaining rules. An example of incompleteness is a totally isolated object. Inconsistency rules are activated whenever there is a change to the model. An example of an inconsistency is that the max is less than the min in cardinality constraints [IPCH91].

[KUNG89] describes a conceptual model that uses a visual and formal approach that models static and dynamic aspects in one model. This conceptual model can incrementally describe the information, has a mathematical basis, and produces an executable specification that can be translated into Prolog. There is no discussion on how the information is entered into the system. Once the information is in, the system evaluates the model. The static information is modeled in an entity-relationship-like language. The dynamic information is modeled in expanded data flow diagrams (DFDs). The elementary processes are modeled similar to Petri nets. Each elementary process has a pre- and post-condition. This format allows for formal analysis of liveness, invariance, and some aspects of correctness. It is different from the object-oriented model because in this model entities are passive components and the separate processes are active components.

The Analyst Assist (AA) project is a knowledge-based tool environment to assist developers in constructing and maintaining a requirements specification. It assists the

analyst in capturing informal requirements, improving the transition from informal to formal requirements, specifying and documenting the requirements using the Jackson Software Development (JSD) method, and validating the specification using prototyping and animation. The initial process of eliciting information is machine-assisted through the use of checklists and facilities for recording the user's answers. A user fact base is created using a fact input tool. The tool is guided by a formulator, which makes use of the domain knowledge base and the current state of the user fact base. Using the information from the checklists, user fact-base, and method and domain knowledge, JSD models are built and presented for checking [LOUC88] [LOUC90].

The Programmer's Apprentice project is studying how software engineers analyze, modify, specify, verify, and document software systems and how these tasks can be automated. The near-term goal is the development of a Programmer's Apprentice, which can act as a software engineer's partner and critic, taking over simple tasks and helping with more complex ones. Part of this project is the Requirements Apprentice (RA), which assists the analyst in the creation and modification of software requirements. The RA does not interact with the end user, but is an assistant to the analyst. Therefore it does not have to deal with natural language input and can use a more restrictive command language. The RA produces as output conclusions drawn and inconsistencies found, a machine-manipulable knowledge base that contains everything known about an evolving requirement, and a requirements document summarizing the knowledge base. The RA has three components. Cake is a knowledge representation and reasoning system that supports propositional deduction and equality reasoning. Cake also maintains dependencies between deduced facts and the incremental retraction of facts. There is an executive that handles user interaction. The last component is a cliché library that contains information on requirements in general and on domains of interest. The clichés are the heart of the system and allow the RA to critique what is in the requirements as well

as what is missing. Clichés are a way of representing, organizing, and applying domain knowledge. They represent commonly occurring structures in the domain. A major research goal of the RA is the codification of the clichés. Every requirement entered into the RA has to be recognized as an instance of some existing cliché. New domains are covered by defining new clichés. A cliché consists of a set of roles and constraints between them. The roles are the parts that vary from one use of the cliché to another. The constraints specify how the roles interact, and they place limits on the parts that can be used to fill the roles. The clichés are organized hierarchically and are represented as frames. These frames are linked by constraints and arranged in an inheritance library. Roles are represented by slots, and the constraints are predicates on the slots. Incompleteness in the specification is handled by making sure that all necessary roles (slots) of the cliché are filled in. An example of a cliché is a repository with roles of collection, patrons, staff, repository additions, and repository deletions. The user inputs a Lisp expression whose first component indicates the type of the command [REUB91].

Kibitzer was created to address the stages of problem identification and conceptualization in knowledge based system design. It helps an analyst create a model of the problem domain that consists of concepts and relations. The domain model is encoded in MetaClass, an object-oriented environment for Common Lisp. The Kibitzer system monitors the editing commands and formulates suggestions and warnings regarding the course of model development. The user interface is graphical using multiple windows. The user input is constrained English and menu-type suggestions. The concepts (like classes) are defined, and the relationships between the concepts are identified by the user of Kibitzer. The Kibitzer system uses inheritance. Based on how the concepts and relationships are named, Kibitzer can tell if a concept is a subclass of another concept and what classes relationships belong to. Kibitzer uses a library of clichés for domain knowledge that are very much like those of the RA [SCHO91].

STES (specification transformation expert system) is a system that translates a requirements specification expressed in terms of DFDs into a design specification in terms of structure charts. STES contains a knowledge base that contains information on the structured design methodology and heuristic guidelines to help determine when certain methods should be applied [TSAI88].

The KIT-LERNER project is based on the use of sloppy modeling. The BLIP system, which is part of the KIT-LERNER project, is a knowledge acquisition system designed to acquire basic problem-solving independent knowledge about a domain, including terminology and simple empirical knowledge. The user enters facts, predicates, and rules in a windowed environment. There are no prestructured activity sequences; any of the operations are available to the user at any time. The BLIP system is implemented in Prolog and Lisp [WROB88].

ESA (Expert Supported OOA Tool) is a system based on an extension of Coad and Yourdan's *OOA methodology*. *ESA is a graphical environment* that provides icons for OOA structures and a limited knowledge-based analysis to critique the completed OORA model. There is no assistance for creating the model [SCHA92].

The KBRAS system is intended to automate the process of acquiring software requirements by providing eliciting and modeling facilities. KBRAS's conceptual models consist of an environmental and virtual model. The environmental model defines objects, object types, attributes, relationships, and associated constraints. The virtual model consists of objectives, activities and states for achieving those objectives, and the control knowledge that describes the software behavior. The KBRAS user interface is graphical and accepts restricted natural language. It contains knowledge on the form of the representation of a domain, the process for how the domain specific knowledge is acquired, combined, and used to model a system, the heuristics used to guide and optimize the integration of system components, and the knowledge required for checking for

inconsistencies and conflicts. The requirements acquisition knowledge is a set of procedures, and the domain specific knowledge is represented by rules and frames. The internal notation produced as output is not formal [ZERO91].

Systems Using Formal Language as Input and Representation of the Specification.

Kate takes as input a formal specification and a context in which to analyze it and outputs a critique which consists of textual reports and simulation. Kate is a computer-based critic. The formal language has the expressive power of Petri nets, plus it adds a class hierarchy, place capacities, and the ability to associate computable predicates with arcs. The critic contains a model of the domain, a matcher to connect the model with the input specification, and a critiquer that does the analysis. The model has a set of policy issues for building systems in the domain and relevant cases. Policy issues are potential specification goals. These represent past experiences in a domain in the form of cases and *scenarios*. The cases are operational, and they provide an abstract behavioral description that ties the nonoperational policy issues with the concrete behavior in the specification. The critic identifies policies that are not supported, are obstructed, or are not necessary [FICK88].

FRORL (Frame and Rule Oriented Requirements Language) is used with a method called the predominance/particular method and a knowledge base to support the requirements acquisition process. FRORL uses frames for object-oriented modeling and production rules for specifying actions and constraints of the domain. Abstract relationships used in FRORL are is_a (instantiation), a_part_of (whole/part), and a_kind_of (inheritance). The predominance/particular method emphasizes that the main features should be represented first using simple and general descriptions; details are then added incrementally. The knowledge base contains rules for specification evaluation, prototype validation, and the translation of the specification into Prolog code. The

production rules are in Prolog. The input to the system is a set of frames. FRORL starts by describing the main system features using frames and then adds detail. The specification is executed in order to validate it. A query system allows the user to ask questions about the evolving specification. FRORL supports default and multiple inheritance. Two types of frames are input: object frames and activity frames. Object frames contain information on the relationship with other object frames (is_a, a_part_of, a_kind_of), the attribute names and the associated attribute values. Activity frames represent changes taking place in the domain. They have five slots: the abstract relation (usually a_part_of), part (the objects or attributes taking part in the activity), precondition for the activity, action (if the precondition is met, do the action), and alt_action (if the precondition is not met, do the alt_action). The parameter list of the activity is the set of parts. To allow the specification to be incrementally built, nonterminal symbols are used to express a term that will be precisely defined later in the development of the specification within FRORL. The knowledge base contains knowledge for checking the specification for proper syntax and consistency, prototype validation knowledge that is used to execute the specification and answer queries, and transformation knowledge that produces Prolog code from the specification [TSAI89].

HCLIE (Horn Clause Logic with Inheritance and Exception) is a language that is a superset of ordinary Prolog. It adds the syntactic category of common nouns. Common nouns are distinguished from predicates by the prefix "kind". HCLIE allows default inheritance, which is where inherited properties can be overridden. It also allows multiple inheritance [TSAI91].

Relationship of Technology Areas to OAKS

Overview. OAKS is built around an object-oriented domain model that is modified to produce an object-oriented model for a particular problem in that domain. The OAKS system is based on OORA principles, structures, methods, and guidelines and rules. Only object-oriented structures, relationships, and techniques were used to develop the OAKS system, in contrast with using other techniques such as function-oriented or dynamic-oriented.

Requirements Analysis. OAKS produces a requirements specification in the form of an object-oriented model of the problem to be solved, which is referred to as the problem model. The problem model of OAKS is the conceptual model of the system to be developed. The OAKS evolving problem model is in informal form, as inconsistencies and incompleteness are allowed. These are removed before the problem model is considered completed.

The main purpose of OAKS is to manage the requirements analysis process. OAKS contains knowledge of a process that is used to elicit requirements from the user, build the problem model, recognize an incomplete and inconsistent problem model, and display the evolving problem model to the user for verification. The OAKS problem model adheres to the concept of a specification in that it specifies what the required system that is to do, but not how to do it.

OAKS represents the specification using a formalized representation consisting of CLOS and LISP code. This representation can be used in different and varying domains.

OAKS does not force a certain sequence of decisions on the user. Instead, it allows the user great flexibility in choosing the sequence of changes made to the problem specification.

OAKS is tolerant of incompleteness, and is easily augmentable.

Future work in OAKS should examine the transformation of the problem model into a formal specification that could be used to develop code.

Requirements Analysis and Artificial Intelligence. OAKS contains knowledge of the domain of interest, the OORA process, the structure of the model of the OORA specification, techniques for transforming the domain model into the problem model, and the process of eliciting information from the user.

OAKS's role is closest to that of "sloppy" modeling [WROB88]. The user is not required to develop a complete problem model before interacting with OAKS. There is a minimum order imposed on the changes that can be made to the evolving problem model. OAKS tries, where possible, to organize and complete changes entered by the user. OAKS tries to maintain integrity and consistency of the problem model and provides immediate feedback on the consequences of all changes. OAKS allows the user to view the problem model at any time and at various levels.

OODA. The OAKS domain model contains the objects, operations, and relationships in a problem domain so they can be modified to produce different problem models in the domain. An OODA must be conducted to produce the domain model. This research did not, however, address the process of OODA. It addressed the form, use, and changes of a domain model, once it is created through the OODA process. OAKS does not provide the tools or guidance for conducting the OODA. This would be done by an analyst and the results encoded in OAKS prior to a user's interacting with OAKS.

OORA. OAKS conducts an OORA by creating a problem model that represents an object-oriented specification.

OAKS does not copy any one OORA model or process. Instead, it borrows components, relationships, steps and guidelines from numerous OORA processes. Therefore, the first step in developing the OAKS system was the development of a math-

based model containing the components and relationships that would be used in OAKS. This step also developed the terminology and definitions used in the OAKS system. The second step evaluated existing OORA processes to gather guidelines and rules that could be used in the OAKS domain model and in the evaluation of the evolving problem model.

Existing Systems. OAKS contains all the components and relationships of an object-oriented specification in their object-oriented form. That is, classes are the central, active components that encapsulate their attributes and services. OAKS also does more than provide a set of structures from which to construct a specification. OAKS contains knowledge of the OORA process to guide in the development of a problem model that is consistent and complete with respect to the defined object-oriented guidelines and rules and the structure of the OAKS domain and problem models. These two characteristics of OAKS make it unique among the existing automated systems.

The existing systems either do not express an object-oriented specification in a true object-oriented form or they provide little guidance and contain little if any knowledge of the OORA process. The systems using frame-based techniques encapsulate attributes within the classes, but services are separate, active entities that are only referred to within the class. Those that produce a true object-oriented specification provide graphical support and limited checking but provide little process support or knowledge. The Analyst Assist and the KBRAS projects come closest to the concept of OAKS, except that neither produces an object-oriented specification. Some, like the Requirements Apprentice, are centered around a particular domain of interest, but they contain little OORA process or model knowledge.

OAKS does not currently contain a natural language front-end, although such a front-end would be ultimately desirable. It also does not require a formal specification as input. This enables the user of OAKS to interact in a way more natural for this phase of software development.

OAKS is the only knowledge-based system that was started from a basis of an OORA system and built around this defined system. The development of OAKS placed importance on a pure object-oriented approach to the model components, relationships, and the evaluation of the evolving model.

III. Methodology

Overview

The main objective of this research was to investigate the feasibility of a computer-based system that can guide a user of the system in the conduct of an OORA resulting in an object-oriented specification of the system to be developed. The proof-of-concept system developed is called the OORA Automated Knowledge System (OAKS). OAKS contains a domain model that serves as a system template, and it is modified by the user to produce a model of a particular system.

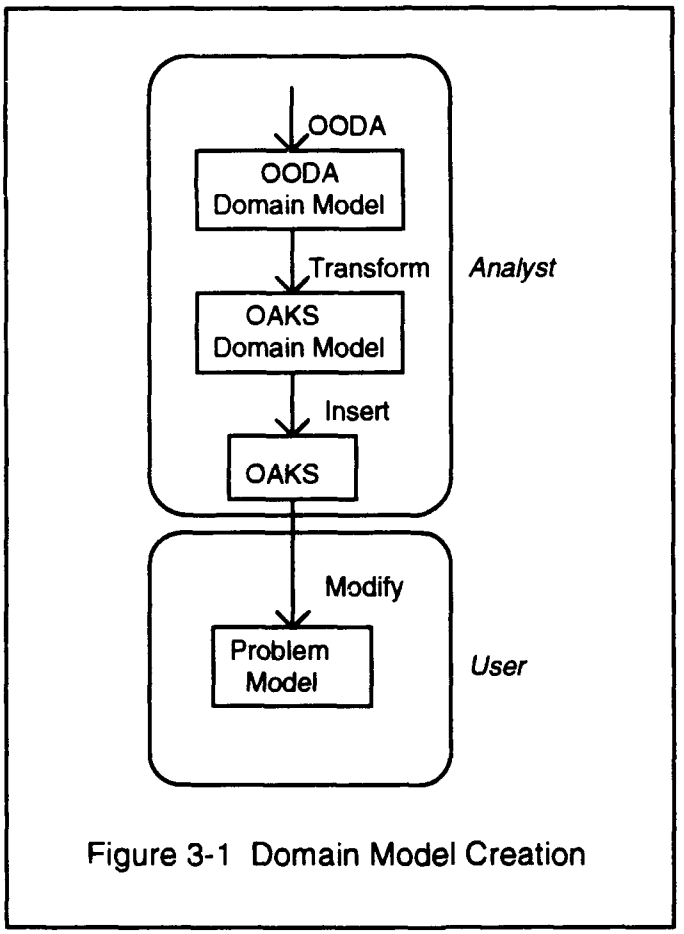


Figure 3-1 Domain Model Creation

The domain model is developed by an analyst who is familiar with a domain and with OORA. Figure 3-1 shows the steps involved in creating a domain model for OAKS. The analyst first conducts an OODA in a domain. This results in a generic set of classes and relationships for that domain. The analyst then transforms the OODA results into the code structure used for the OAKS domain model. This code is inserted into OAKS and tested to ensure it meets OAKS guidelines and rules. After the tests have been successfully completed, OAKS is ready for the user to modify the domain model and create a problem model in that domain. The research addresses the basis and structure of OAKS, but does not address the OODA process used to create the domain model.

OAKS was developed in six steps. Each of the steps built on the findings of the previous step(s). The first step defined an OORA mathematical model which contains the components of the OORA model and their relationships. All order constraints on acquisition of the components were defined. The second step defined how to acquire and evaluate the components of the OORA model. The evaluation information took the form of guidelines and rules. This provided information on how to evaluate the model and the model component interactions. The third step defined how to implement the guidelines and rules developed in step two for the purpose of evaluating the object-oriented requirements model. Step four prototyped OAKS. The prototyping of the system required the development of a code structure that represents the domain model, a component that analyzed the domain and evolving problem model, a component that modified the domain model to produce the problem model, and a user interface. The fifth step tested OAKS, and the sixth analyzed the results. Figure 3-2 shows all the steps and their order.

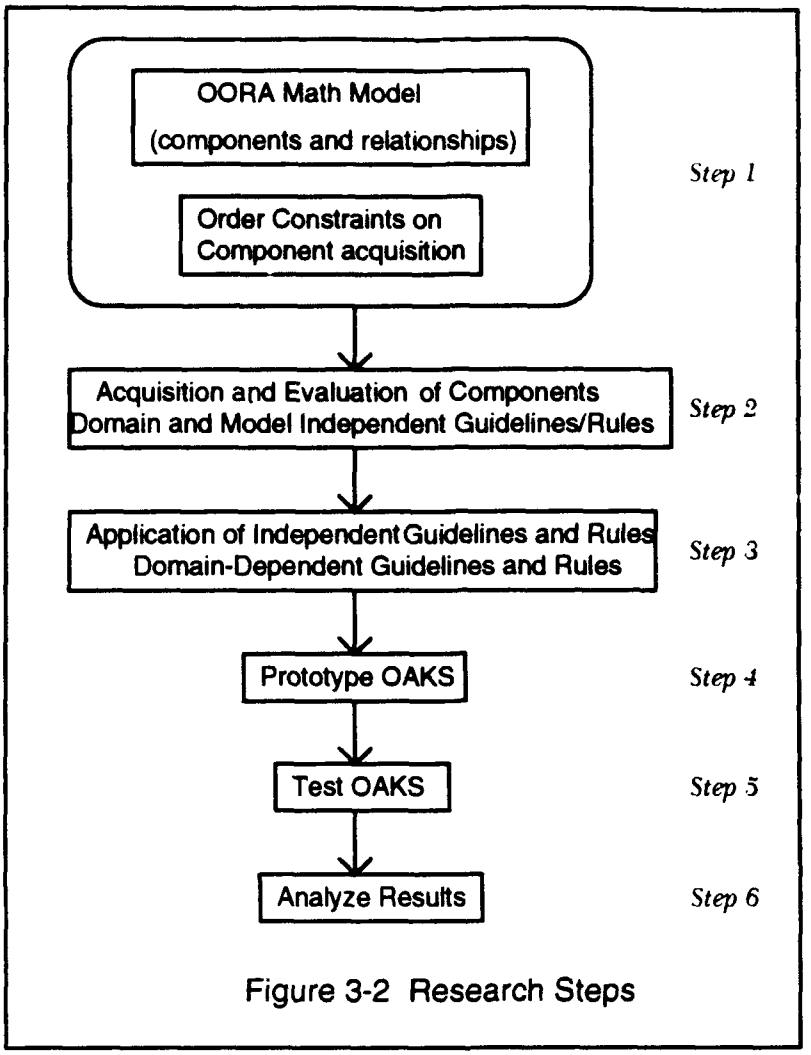


Figure 3-2 Research Steps

Step 1. Define an OORA Mathematical Model

The OORA mathematical model is one that contains the components and relationships necessary for the development of an object-oriented requirements model of the system to be developed. The system to be developed is the solution to a particular problem in a domain, called the "problem model".

The OORA mathematical model was developed by analyzing existing OORA processes. These processes were used as a basis for a set of components and their

relationships that are necessary for an object-oriented specification. These components were used as the components of the OORA mathematical model. This mathematical model was used as the basis for the domain model in OAKS.

Step 1 produced information on what, for the purposes of this research, was considered a "good" OORA model and the required relationships between the components of the model. This information was required to enable OAKS to evaluate the completeness and consistency of the domain model and the evolving problem model as far as the inclusion and form of the necessary components.

This step enumerated all the components necessary for an OORA and the resulting specification, their allowed relationships and the order in which they are acquired. Another product of this step was a rationale on why the components and relationships of the resultant OORA model were selected and why certain components of existing OORA models were included or excluded.

Step 2. Define Acquisition and Evaluation of the OORA Model

Step 1 defined the components, their relationships, and the order of acquisition of the components. Step 2 collected domain and model independent guidelines and rules on how to acquire and evaluate those components and relationships to ensure they meet OORA constraints, rules and guidance. These guidelines and rules were collected by analyzing existing OORA processes and extracting guidelines and rules on the creation, form and changes of components of the OORA model. The guidelines are those criteria that are suggested but not required. An example of a guideline is that a class should have more than one service. A rule is a criteria that must be adhered to. An example of a rule is that a subclass must inherit all the attributes and services of its superclass(es). The guidelines and rules on acquiring the model components were used in the development of the OAKS

domain model. Some of the guidelines and rules for evaluating the model components were embodied in the OAKS model and some were quantified in step 3 so they could be applied by OAKS to the domain and problem models.

These guidelines and rules that were based on existing processes were in a form that was informal and not directly usable by a computer-based system. Furthermore, they did not form a complete set. These guidelines and rules were gathered from a review of the literature. In Step 3, these guidelines and rules were refined and further guidelines and rules are added based on the domain information. Step 4 developed guidelines and rules that were based on the OAKS domain and problem model structure, and the OAKS model modification process.

The guidelines and rules analyzed in this step were independent of the domain and of the OAKS model structure. They were developed based on the desired static characteristics of the OORA model. Their purpose is to evaluate the model components, the interactions between the model components, and the entire model. These domain-independent guidelines and rules are applied to any domain used in OAKS, because they are based on the desired characteristics of any OORA model.

Step 3. Define Application of Guidelines and Rules

Step 2 developed a set of domain-independent guidelines and rules that were applicable to any OORA model. However, these guidelines and rules were not in a form that could be used by a computerized system. Many were too subjective, using words such as "best" and "may be". These guidelines and rules also did not take into account the domain of interest or the structure and process of the implemented OAKS system. Guidelines and rules based on the domain of interest are discussed in this step. Guidelines and rules based on OAKS structure are discussed in Step 4.

In this step, the guidelines and rules of Step 2 were examined to determine which would be used to evaluate the domain and problem model and how they would be used. Some of the guidelines and rules were used in the development of the domain model but not used to evaluate the model once it was in OAKS. For example, standard terminology for the domain was used to create the class names in OAKS. A user manual for OAKS would contain suggestions to use standard terminology when creating a new class in OAKS, but the OAKS system itself cannot check for standard terminology.

The form of domain-dependent guidelines and rules was also developed during this step. Domain-dependent guidelines and rules provide a more complete analysis of the domain model. This information is used by the OAKS system to ensure consistency and completeness of a model in a particular domain. An example of a domain-dependent rule is that a certain class in a domain cannot be deleted because it is essential in that domain.

A particular domain was chosen in this step. The chosen domain was that of a system that manages the scheduling and maintenance and flights for an Air Force aircraft squadron.

Step 4. Prototype OAKS

The prototyping of OAKS was required as a proof-of-concept of the feasibility of a computer-based system that guides the OORA process. OAKS required the design and implementation of code structures to represent the OORA model and the relationships between components in the model. The code structure contained all the components and relationships in a form that could be used in any domain in which an OODA could be performed. The model had to be flexible yet accessible, so that the structure could be constantly checked during the user modification process.

After the structure of the OAKS domain model was completed, the structures in code that analyzed both the domain and problem model in accordance with the guidelines and rules established in step 3 were developed. The analysis code was used both to check the validity of the domain model prior to its use, and to check the validity of the evolving problem model.

The development of code structures that would enable the modification of the domain model by the user followed the development of the analysis code because the analysis code was used to evaluate all changes. This code controlled the modification process so that the model remained consistent and complete in accordance with model structure and OORA guidelines and rules. It also controlled the order in which changes were made, if such an order was necessary to maintaining a valid model.

The last code developed was the graphical user interface. This allowed easier access to the model and the modification process. OAKS uses the domain model as an initial template and then guides the user through the creation of a problem model through a series of refinements to the domain model. The refinements can be done in any order, except for the requirements for order based on the model itself, and previous refinements can be changed at any time.

Step 5. Test the OAKS Prototype

The testing of the OAKS prototype was an integral part of the development of the OAKS code. As each LISP procedure was developed, it was tested as a separate entity to ensure proper operation. The LISP procedures were then grouped into functional areas of code, such as a set of procedures that checked the proper structure of an attribute, and then tested again. Each of the files that make up OAKS was then tested. First the file "oaksd:lisp", which contains the OAKS model structure and the domain model, was

developed and tested. Next, the model evaluation code in "oaksno.lisp" was developed and tested. This required the use of "oaksd.lisp". The OAKS modification procedures in "oaksmod.lisp" were developed and tested next. These required the use of the files "oaksd.lisp" and "oaksno.lisp". The user interface (UI) was developed next. The UI uses three files: "oaksui.lisp", which contained the LISPView code that creates the windowed user interface; "oaksave.lisp" which saves the evolving domain model to a file and retrieves it for any OAKS session; and "oaks.lisp" which loads all of the above files and created the environment for using OAKS. The UI code is dependent on code from "oaksd.lisp", "oaksno.lisp" and "oaksmod.lisp". Therefore, the development of OAKS followed a building block approach, with each procedure building a file, which was used in the development of the next file.

Step 6. Analyze the Results

The results were analyzed for future directions, problems, usefulness, areas that need further investigation, and overall results. Given the size of the task, an important part of the analysis was what work is left to be done and how it fits into the existing OAKS system. OAKS was developed as a proof-of-concept system, whose emphasis was on the development of a system that adhered as closely as possible to object-oriented concepts and contained knowledge of OORA process and principles to guide in specification development. Areas such as the user interface, dynamic characteristics of the model and translation to design and code were de-emphasized.

IV. OORA Mathematical Model

General

An OORA model consists of a set of classes, denoted as Set_Of_Classes, and the relationships among the classes. These relationships are the inheritance relation which consists of the superclass and the ancestor relation, the whole/part relation, and other relations. The OORA model will be discussed by describing all of its components and building these components into a full model.

A class consists of a name, a set of attributes, and a set of services. The services include how to create and destroy objects of that class.

Class Attributes

Each class contains a set of attributes. Each attribute is identified by a name and a set that defines all possible values that attribute can be assigned. For example, for the class "Flight-Schedule" there may be an attribute named "Take-Off-Time" whose legal set of values is the set of integers between 0 and 2400 (using military time). Each object contains its own value for each of the attributes of the class for which it is an instance. Therefore, the value of an attribute can only be determined from knowledge of the object identification. For example, if there is a flight schedule "Schedule-A" that is an instance of the class "Flight-Schedule", the attribute "Take-Off-Time" may take on the value "1300" for the object Schedule-A. For another object, the attribute may take on another value from the set of legal values. The values of all the attributes of an object make up the object state.

The name of each attribute, **Attr_Name_i**, can be modeled as a function. The domain of an attribute function is the set of all objects that are instances of the class to which the attribute belongs. The set of all objects of a class named **Class_Name**, denoted by **OBJ(Class_Name)**, is defined as shown in relation (1).

$$\text{OBJ(Class\_Name)} \equiv \{\text{obj} \mid \text{obj } \textit{is an instance of } \text{Class\_Name}\} \quad (1)$$

The range of an attribute function is a set of legal values for the attribute, **Range_i**. The function associated with each attribute name takes as input the name of an object and returns the value of the attribute for that object. The returned value must be a member of the set of legal values for the attribute, Range_i. In particular, Attr_Name_i maps values from OBJ(Class_Name) into Range_i.

$$\text{Attr\_Name\_i} : \text{OBJ(Class\_Name)} \rightarrow \text{Range\_i} \quad (2)$$

where Attr_Name_i is the name of an attribute within Class_Name, OBJ(Class_Name) is the set of objects that are instances of Class_Name, and Range_i is the set of legal values for Attr_Name_i.

The class state space, **Class_State_Space**, can be represented as a finite set of pairs consisting of attribute names and their ranges.

$$\text{Class\_State\_Space} \equiv \{(\text{Attr\_Name\_1}, \text{Range\_1}), .. , (\text{Attr\_Name\_n}, \text{Range\_n})\} \quad (3)$$

where Class_State_Space is the state space of a class, Attr_Name_1 and Attr_Name_n are attribute names of that class, and Range_1 and Range_n are the legal values of Attr_Name_1 and Attr_Name_n respectively.

If there are default values for the attributes, these are shown within the CREATE service for the class in the postcondition. The CREATE service creates a new object of that class with any desired default values of the attributes.

Class Services

Each class contains a set of services that implement the behavior of the class. The service names, denoted by **Service_Name,** can be represented as functions whose domain consists of an object state, denoted by **Object_State**, and a list that represents an optional input parameter list, **Input_List**, and whose range is the possibly changed object state, **Object_State**, and a list that represents an optional output parameter list, **Output_List**.

Object_State is a set that consist of the values of all the attributes of the object.

$$\text{Object\_State} = \{\text{Attr\_Name\_1 (Object\_Name)}, \ldots, \text{Attr\_Name\_n (Object\_Name)}\} \quad (4)$$

where Object_State is the set that consists of the values of the attributes of the class, Attr_Name_i is the name of an attribute of the class, and Object_Name is the name of an object of the class.

Input_List is a list of sets that represent the legal values that bound each of the required input parameters of the service. Output_List is a list of sets that bound each of the output values of the service.

$$\text{Input\_List} = \langle \text{Input\_Set\_1}, \ldots, \text{Input\_Set\_m} \rangle \quad (5)$$

$$\text{Output\_List} = \langle \text{Output\_Set\_1}, \ldots, \text{Output\_Set\_x} \rangle \quad (6)$$

where Input_Set_i is the set of legal values for one input parameter and Output_Set_i is the set of legal values for one output parameter.

Therefore, each service name can be represented as:

$$\text{Service\_Name} : \text{Object\_State} \times \text{Input\_List} \rightarrow \text{Object\_State} \times \text{Output\_List} \quad (7)$$

where Service_Name is the name of the service, Object_State is the set that consists of the values of the attributes of a class before the service is executed, Input_List is a list of sets that represents the legal values for the input parameters, Object_State is the set that represents the values of the attributes of the class after the service is executed, and Output_List is a list of sets that represents the legal values for the output parameters. The symbol **X** represents the cross product.

The values making up the object state are accessed by the use of the attribute names, a *function*, as described in the previous section.

For example, the class of Dog contains an attribute License_Number, which is an integer, and a service Change_License_Number that takes a new license number and replaces the old one. An instance of the class of Dog is Fido. Object_State would consist of the value of the one attribute, which in this case would be {License_Number(Fido)}. The input list is a value from the set of integers that represents the new license number. The output list is empty in this example.

In addition to specifying a representation for the service name, the operation of the service must also be specified. The operation of the service can be specified by using a precondition and a postcondition. The precondition, denoted by **Pre**, is a predicate that represents assumptions on the object state (Object_State) prior to the execution of the service, and the assumptions on the values of the input parameters in Input_List. The postcondition, denoted by **Post**, is also a predicate that represents the required relationship

between the input values, which consists of the Object_State and Input_List, and the output values consisting of the Object_State and Output_List.

Hence, given relationship (7), then:

$$\text{Pre}(s, i1, .., im) \rightarrow \{\text{True, False}\} \quad (8)$$

$$\text{Post}(s, i1, .., im, s', o1, .., ox) \rightarrow \{\text{True, False}\} \quad (9)$$

where:

Pre is the precondition

Post is the postcondition

$s \in$ Object_State

$(i1 \in$ Input_Set_1$) \Lambda ... \Lambda (im \in$ Input_Set_m$)$

$s' \in$ Object_State

$(o1 \in$ Output_Set_1$) \Lambda ... \Lambda (ox \in$ Output_Set_x$)$

If the input is in the domain and the precondition is true, then the postcondition is implied. If the input is not in the domain or the precondition is false, nothing is known about the output values of the service.

If the services of other classes are required to satisfy the requirements of the service, the postcondition representation must contain references to those services. The form of the reference to services of other classes is dependent on the form of the postcondition. In this research, the services of other classes are represented by using a dot notation, where the class name is separated from the service name using a dot, such as in "Class_A.Service_A". For example, the class Squadron may have a service called "Number-Operational" that returns the number of aircraft in the squadron that are fully mission capable. This would require a message from Squadron to each Aircraft object in

the squadron asking its status. This message would refer to the service of the Aircraft class, called "Get-Status". By using the dot notation, the "Get_Status" service of an object of the aircraft class is represented in a postcondition of the "Number_Operational" service by "Aircraft.Get_Status".

When the operation of a service is specified, it is assumed that the service will be operating on the state of a object that is an instance of the class to which that service belongs. Therefore, the information required when specifying a service denoted as **Service_i**, is the service name Service_Name (which is a function name), the input parameters Input_List, the output parameters Output_List, the precondition Pre, and the postcondition Post. Both the input parameters and the output parameters are optional, and the precondition may be "true", signifying that any input state is acceptable. All input and output values are assumed to be members of the corresponding input and output sets.

Therefore, a service can be specified using the following tuple:

$$\text{Service\_i} \equiv (\text{Service\_Name\_i}, \text{Input\_List\_i}, \text{Pre\_i}, \text{Output\_List\_i}, \text{Post\_i}) \qquad (10)$$

where Service_i is a service, Service_Name_i is the name of Service_i as defined by relationship (7), Input_List_i is the input parameter list as defined by relationship (5), Pre_i is the precondition as defined by relationship (8), Output_List_i is the output parameters as defined by relationship (6), and Post_i is the postcondition as defined by relationship (9).

The set of all services of a class, denoted by **Services**, is denoted by:

$$\text{Services} \equiv \{\text{Service\_1}, .. , \text{Service\_s}\} \qquad (11)$$

where Services is the set of all services of a class and Service_i is one service of the class as defined by relationship (10).

The preconditions and postconditions of the services of a class reflect all the possible states of the class. The preconditions show all possible states prior to service execution and the postconditions after service execution. The CREATE service shows the initial state of an object of a class through its postcondition. Any service whose precondition is true in the initial state of the object can be executed in that initial state, and the postcondition shows the new state. Therefore, all possible states are specified by following all possible service execution paths through the lifecycle of an object.

In reality, the only states that are of interest are those that are specified in the preconditions of the services. For example, there may be a service called "Change-Squadron" that changes the Squadron attribute of the Aircrew class discussed above. The precondition for this service will most likely be "true", because any squadron will be replaced by the new squadron. Therefore, even though there is a change in the state of the object, it is not a state change that would allow different services to execute or not execute because the value of the preconditions changed. On the other hand, going back to the Aircraft class example, if there existed a service called "Schedule-Recheck", a likely precondition would be that the aircraft already had the initial check of the system in question. Therefore, two states of interest would be (1) the aircraft has not had its initial check, and (2) the aircraft has had its initial check. This state would be changed by a service such as "Conduct-Initial-Check".

Certain states of a class determine how a service operates and therefore constitute the states of interest for a class. All possible states of interest of a class are not mutually exclusive. An object may simultaneously exist in more than one state of interest. The precondition of a service may use information on the status of none, one or more states of interest to determine its true or false value. For example, assume there is a difference in

how a service or services in the Aircraft class operate based on the value of the Status attribute and a difference on how a service or services operate based on whether the aircraft has had its initial check of a system. Therefore, an aircraft can simultaneously exist in two states of interest: the state of interest of being fully-mission-capable and the state of interest of not having had the initial check.

Classes

The identification of classes by name is a key component of OORA. Object names are used to find classes by looking at how the objects may be grouped into classes, but the objects are not used in the final model. The emphasis in OORA is the identification of classes. The identification of the specific objects is done during design. The classes represent the structure of the objects of the system. That is why Coad and Yourdan *[COYO91] call the structures* created during analysis "class-&-objects". It is not necessary during analysis to identify all objects that may be created by a particular class. It is only necessary to identify all problem space classes by name. Solution space classes will be identified during design.

Classes consist of the class name, denoted by **Class_Name**, the set of class services, denoted by Services, and the state space or the set of attributes, denoted by Class_State_Space. Services were defined in section 3.3 and Class_State_Space in section 3.2. Therefore, each class, denoted by **Class_i**, can be represented by the tuple:

$$\text{Class\_i} \equiv (\text{Class\_Name\_i}, \text{Class\_State\_Space\_i}, \text{Services\_i}) \qquad (12)$$

where Class_i is a particular class, Class_Name_i is the name of the class, Class_State_Space_i is the state space as defined by relationship (3), and Services_i are the services of the class as defined by relationship (11).

All the classes in an OORA model can be represented as a set denoted as **Set_Of_Classes**:

$$\text{Set\_Of\_Classes} \equiv \{\text{Class\_1}, \ldots, \text{Class\_n}\} \tag{13}$$

where Set_Of_Classes is all the classes in the OORA model and Class_i is one class as defined by relationship (12).

Inheritance Relationship

The inheritance structure is a key element of object-oriented analysis results. All OORA methods include this structure in some form, mostly represented as a digraph.

The inheritance relationship can be described as a binary mathematical relation. Let A and B be sets. The Cartesian cross product of A and B is defined by:

$$A \mathbf{X} B = \{\langle a,b\rangle \mid a \in A, b \in B\} \tag{14}$$

A binary relation R is a subset of A X B, where A is the domain of R and B is the codomain.

$$\langle a,b\rangle \in R \Leftrightarrow aRb \tag{15}$$

Therefore, the inheritance relationship, denoted by $\mathbf{R_I}$, is a binary relation on the set of classes in the system, denoted as Set_Of_Classes. That is,

$$R_I \equiv \{<a,b> | a,b \in \text{Set\_Of\_Classes} \wedge a \textit{ is the immediate parent of } b\} \qquad (16)$$

where R_I is the inheritance relationship and Set_Of_Classes is the set of all classes in the model as defined by relationship (13).

This inheritance relation R_I is called the **superclass** relation, where the superclass is the immediate parent of each class. R_I has the following characteristics:

(i) R_I is irreflexive, that is, $<x,x> \notin R_I$, $\forall x \in$ Set_Of_Classes.

(ii) R_I is antisymmetric, that is, $(x\ R_I\ y \ \wedge\ y\ R_I\ x) \Rightarrow (x = y)$, $\forall\ x,y \in$ Set_Of_Classes.

The **ancestor** relation, which describes all classes from which a class inherits, is described by the transitive closure of R_I, denoted by $t(R_I)$. The transitive closure of R_I is the relation $t(R_I)$ such that:

(i) $t(R_I)$ is transitive.

(ii) $t(R_I) \supset R_I$.

(iii) For any transitive relation $t(t(R_I))$, if $t(t(R_I)) \supset R_I$, then $t(t(R_I)) \supset t(R_I)$.

Therefore, if R_I is a binary relation on A, then $<a,b> \in t(R_I)$ iff there is a sequence of elements $<c_0, c_1, .. , c_n>$, $c_i \in$ A, where $n \geq 1$, $c_0 = a$ and $c_n = b$, and for $0 \leq i < n$, $<c_i,c_{i+1}> \in R_I$.

Both the superclass relation, R_I, and the ancestor relation, $t(R_I)$, can be represented as digraphs. An inheritance digraph D_I is an ordered pair $D_I = <A,R_I>$ where A is the set of vertices and R_I is the superclass binary relation on Set_Of_Classes. The elements of Set_Of_Classes are vertices of D_I. The elements of R_I are the arcs. Also, $<a,b> \in t(R_I)$ iff there exists a path of nonzero length from vertex a to vertex b.

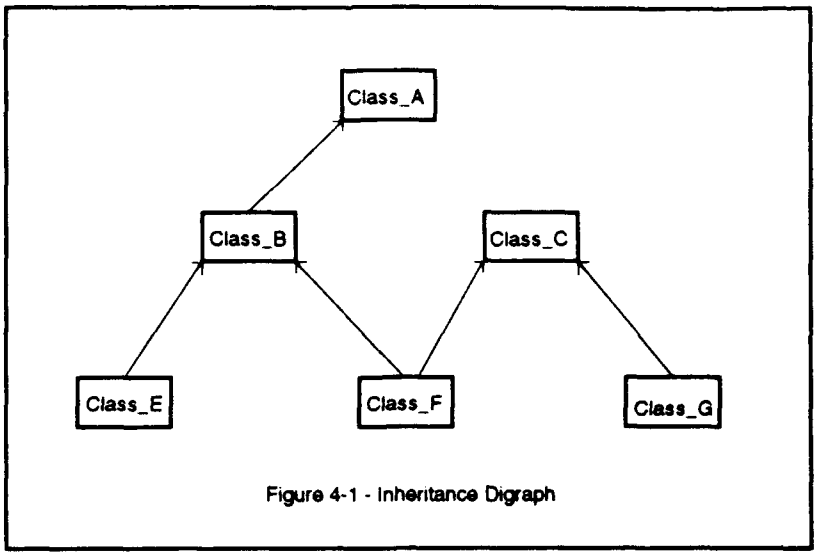


Figure 4-1 - Inheritance Digraph

In Figure 4-1:

R_I = {<Class_A, Class_B>, <Class_B, Class_E>, <Class_B, Class_F>, <Class_C, Class_F>, <Class_C, Class_G>}

$t(R_I) = R_I \cup$ {<Class_A, Class_E>, <Class_A, Class_F>}

Whole/Part Relationship

Some of the OORA methods break out the whole/part relationship separately and some identify it as a named relationship. Because it is a relationship that is widely discussed as an important relationship to model, it is discussed separately here. According to [COYO91], one of the ways to deal with the complexity of the OORA model is by breaking the model into its component parts, and then dealing with each of the parts. Also, looking at classes in the view of what they could be part of and what the components of the classes are helps identify other classes in the problem space.

To model the binary whole/part relation, denoted by $\mathbf{R_W}$;

Let $\mathbf{R}_{\mathcal{N}}$ represent the binary relation on $\mathcal{N}$, the Natural numbers, such that:

$$R_{\mathcal{N}} \equiv \{<a,b> \mid a,b \in \mathcal{N}\} \tag{17}$$

Let $\mathbf{R}_{SC}$ represent the binary relation with Set_Of_Classes as the domain and $R_{\mathcal{N}}$ the codomain such that:

$$R_{SC} \equiv \{<a,b> \mid a \in \text{Set\_Of\_Classes}, b \in R_{\mathcal{N}}\} \tag{18}$$

where Set_Of_Classes is the set of all classes in the model as defined by relationship (13). Therefore, R_W can be represented as a binary relation on R_{SC}:

$$R_W \equiv \{<a,b> \mid a,b \in R_{SC}\} \tag{19}$$

The element of R_{SC} associated with the first element of R_W includes two Natural numbers indicating the least and greatest number of parts that a whole might have at any given moment. The element R_{SC} associated with the second element of R_W includes two Natural numbers indicating the least and greatest number of wholes to which a part may belong.

For example, let w, x, y, z $\in \mathcal{N}$, the Natural numbers, and:

$$<<\text{Class\_1}, <w,x>>, <\text{Class\_2}, <y,z>>> \in R_W \tag{20}$$

Then:

$$<\text{Class\_1}, <w,x>> \in R_{SC} \text{ and } <\text{Class\_2}, <y,z>> \in R_{SC} \tag{21}$$

where:

$$Class\_1, Class\_2 \in Set\_Of\_Classes$$

$$\langle w,x \rangle \in R_N \text{ and } \langle y,z \rangle \in R_N$$

This indicates that Class_1 (the whole) consists of as few as w and as many as x of Class_2 (the part). Also, Class_2 is part of as few as y and as many as z of objects of Class_1.

Other Relationships

Whole/Part relationships are a type of general relationship. Relationships are also called associations [RUM91] and instance connections [COYO91]. Relationships represent a connection between objects. This connection shows objects that are responsible for either representing a concept in the problem or those that together satisfy a responsibility. This connection does NOT represent an inheritance connection or message passing through services.

An example of a relationship is that between an aircraft part and its repair symptoms. An aircraft part has a set of repair symptoms, and there is a set of repair symptoms for each part, but the symptoms are not part of an aircraft part. An aircraft part can have one or more symptoms and a symptom can be associated with more than one part. This relationship is not represented by inheritance, whole/part, or message connection. It should, however, be represented by attributes in one or both classes. For example, the Aircraft-Part class could contain an attribute List-of-Symptoms, that is, the names of all the symptoms that apply to this part. In this case, however, since the relationship is a many-to-many connection, a new class should be created that contains information on what symptoms apply to which aircraft parts.

Relationships are not required to be named. Naming can be confusing because each relationship has two names depending on which object is examined first. For example, an aircraft part "has-a" group of symptoms, but the symptoms "belong-to" aircraft parts. Therefore, the "has-a" relationship would have objects of the class of Aircraft-Part as the domain and objects of the class of Repair-Symptom as the codomain. The "owns" relationship would have Repair-Symptom as the domain and Aircraft-Part as the codomain.

According to [RUM91], most relationships can be represented as binary relations, or can be transformed into binary relations.

Like whole/part relationships, the existence of relationships in the class structure is not known except through the existence of attributes that contain the necessary information describing the relationship. If the knowledge of the relationship is needed for a pair of classes to fulfill some responsibility of the system, then the information in the relationship is needed by one or more services. Services operate on state information, or the attributes. Therefore, the relationship information must be modeled in an attribute or attributes.

[RUMB91] states that a relationship should not be modeled as an attribute within a class. Rather, it should be modeled only as a named relationship. However, if the knowledge of the relationship is needed for a pair of classes to fulfill some responsibility of the system, then the information in the relationship is needed by one or more services. Services operate on state information, or the attributes. Therefore, the relationship information should be named and modeled in an attribute or attributes and not modeled solely as a relationship name within the model.

An obvious question comes to mind. If relationships are represented in attributes or in a new class that contains the attributes that represent that relationship, then why do relationships need to be separately modeled? The answer is that the identification of

associations aids in the identification of attributes and new classes, and in the evaluation of the structure of existing classes. For example, a many-to-many relationship shows that a new class needs to be created. The relationship itself is a model of a concept that will not have a direct translation into code, as message connections do. The identification of relationships documents where certain concepts are represented in the object-oriented model.

Any general relationship can be modeled in the same manner as the whole/part relationship. The relationship must be modeled with the name of both representations. The relationship "has-a/belongs-to" will be used as an example. The same relations used in the whole/part relation can be used to model any general relationship, denoted by $\mathbf{R_R}$:

Let $\mathbf{R_N}$ represent a binary relation on $\mathcal{N}$, the Natural numbers, such that:

$$R_N \equiv \{<a,b> \mid a,b \in \mathcal{N}\} \tag{17}$$

Let $\mathbf{R_{SC}}$ represent a binary relation with Set_Of_Classes as the domain and R_N the codomain such that:

$$R_{SC} \equiv \{<a,b> \mid a \in \text{Set\_Of\_Classes}, b \in R_N\} \tag{18}$$

where Set_Of_Classes is the set of classes in the model as defined by relationship (13).

Therefore, R_R can be represented as a binary relation on R_{SC}:

$$R_R \equiv \{<a,b> \mid a,b \in R_{SC}\} \tag{22}$$

The element R_{SC} associated with the first element of R_R includes two numbers indicating the least and greatest number of objects that are in the relationship at any given

moment in time. The element of R_{SC} associated with the second element of R_R includes two numbers indicating the least and greatest number of objects involved in the relationship.

For example, let R_R represent the "has-a/belongs-to" relationship. Let:

$$\langle\langle\text{Aircraft-Part}, \langle 1,n\rangle\rangle, \langle\text{Repair-Symptom}, \langle 1,m\rangle\rangle\rangle \in R_R \qquad (23)$$

Then:

$$\langle\text{Aircraft-Part}, \langle 1,n\rangle\rangle \in R_{SC} \text{ and } \langle\text{Repair-Symptom}, \langle 1,m\rangle\rangle \in R_{SC} \qquad (24)$$

where:

R_R is the relation "has-a/belongs-to"

Aircraft-Part, Repair-Symptom $\in$ Set_Of_Classes

$\langle 1,n\rangle \in R_N$ and $\langle 1,m\rangle \in R_N$

This indicates that objects of Aircraft-Part can have the relationship "has-a" with as few as 1 and at most n objects of class Repair-Symptom. Also, objects of class Repair-Symptom can have the relationship "belongs-to" with as few as 1 and as many as m objects of class Aircraft-Part.

Constraints

Constraints are functional relationships between classes that restrict the values that the attributes can assume. An example of a constraint would be that an employee may not have a salary higher than that of her boss. Constraints are implemented by the services

that change the values of the attributes. For example, when a salary is changed, the service will need to ensure that the salary is not greater than that of the boss.

Constraints should be identified and the OORA model developed that satisfies those constraints. Constraints are not separate parts of the model but are used to guide development of the model.

Subjects

Subjects, or subsystems, are a way of organizing a model with a large number of classes into groups of classes which form smaller and more manageable pieces. Because this research will only deal with a small problem within the abilities of one analyst, subjects will not be used.

The OORA Model

The entire OORA model, denoted by **OORA_Model**, can be represented as a tuple that consists of the Set_Of_Classes, which contains the classes in the model, the inheritance relation R_I, the ancestor relation $t(R_I)$, the whole/part relation R_W, and other relations R_R:

$$\text{OORA\_Model} \equiv (\text{Set\_Of\_Classes}, R_I, t(R_I), R_w, R_R)$$

where Set_Of_Classes is the set of classes in the model as defined by relationship (13), R_I is the inheritance relation as defined by relationship (16), $t(R_I)$ is the ancestor relation which is the transitive closure of R_I , R_w is the whole/part relation as defined by relationship (19), and R_R represents other relations as defined by relationship (22).

The OORA model contains all the information required for the development of an object-oriented requirements specification. All components discussed in the literature reviewed are represented in this model in some form.

Whole/part relationships are discussed separately from other relationships because they are consistently used in the OORA as a named relationship to help structure the model. The other relationships are problem dependent. They are used when and if the problem requires that type of modeling. Therefore, the process of looking at the model in terms of its wholes and associated parts is consistently used, while other relationships are uncovered as part of the analysis.

When the OORA mathematical model is implemented, the superclasses of each class can be represented as a part of the class structure instead of in a separate inheritance tree. This means that information on the parents of each class is contained in each class. There is an argument for this, because each class must know its superclasses in order to access inherited attributes and services. In the OORA mathematical model, the information on the superclass of each class is contained in R_I, the superclass structure. The same argument applies to not explicitly creating a separate message passing tree structure, whole/part tree structure, or other relationships tree structure. This information can be contained within the class structure itself. In the case of the message passing tree structure, this tree can be derived from the relations for the postconditions of the services of all the classes because they contain information on other classes used.

Ordering of OORA Model Components

There is no strict sequential order for obtaining the elements of the OORA model. Every process discussed in the literature is iterative, but each has a starting point and there are dependencies between elements of the model. Parts of the process can repeat

many times, with information discovered later in the analysis process causing addition and modification of elements of the model.

[COYO91], [RUMB91], [SIBL89], [MONA92], and [NERS92] start with identification of classes. These classes form the basis for all further work on the model. Only [RUBI92] takes a different approach. [RUBI92] focuses on identifying the system behavior first and then defines objects that will exhibit the behavior. The two views are interrelated. Prior to the identification of classes, the analyst must understand thoroughly what the system is to do. This, in essence, is understanding the behavior the system must exhibit. The difference between the two approaches, behavior-first vs. classes-first, is where the emphasis is placed early in the analysis. In the behavior-first approach, the behavior of the system is analyzed and modeled in detail, and then classes are identified that will satisfy the needed behavior. In classes-first, class identification methods are emphasized early in the process so that behaviors can be identified with classes. The system behavior is used as a check to be sure that all necessary classes have been identified.

There are potential problems with the behavior-first approach. First, there is the danger that a more functional, rather than object-oriented, approach will be taken because of the early emphasis on system behavior. This problem can be overcome with careful management of the process. The second potential problem, more relevant to this research, is the potential difficulty of using a domain model as a basis for new, evolving models. If the start of the process identifies behaviors, it will be difficult to map those behaviors to an existing object-oriented domain model, which consists of a class structure. OAKS would have to identify where in the domain model the behavior is (or is not) satisfied. This is much more difficult than determining if a class exists in the domain model. For these two reasons, and because the classes-first approach is more commonly used, the classes-first approach was used in this research.

Even though there is no strict ordering of the steps and the process is iterative, there are dependencies between elements of the model. These dependencies will be shown using the following notation:

Tuple_Name.Tuple_Component

This notation identifies a component of a certain tuple. For example, "Class_Name.Class_State_Space" identifies the Class_State_Space of a certain class, called Class_Name.

The dependencies between a class name and its components can be shown as follows, where $\Rightarrow$ is the symbol for implication:

$\forall$ Class_Name $\in$ Potential_Class_Names,

(Attribute $\in$ Class_Name.Class_State_Space $\Rightarrow$ Class_Name $\in$ Set_Of_Classes) Λ

(Service $\in$ Class_Name.Services $\Rightarrow$ Class_Name $\in$ Set_Of_Classes)

This states that for all classes in a particular model, the class name must exist, prior to the identification of any of its attributes, which make up the state space, or any of its services. This does not state that all class names must be identified before any attributes or services can be identified. It just states that the name of a class must be identified before any of the components of that class. Potential_Class_Names represents all possible class names.

There are relationships between components of the OORA model and the classes as shown in relationships (16), (19), and (22):

Let $R_1 = \{<p_1, c_1>, <p_2, c_2>, .. , <p_n, c_n>\}$

Then $\forall i \in 1 .. n, (p_i \in Set\_Of\_Classes) \Lambda (c_i \in Set\_Of\_Classes)$

This states that the class must exist before it can be used in an inheritance relation, or any other relationship, such as whole/part, as the following shows:

Let $R_W = \{<<a_1, <b_1, c_1>>, <d_1, <e_1, f_1>>>, .. ,<<a_n, <b_n,c_n>>, <d_n, <e_n, f_n>>>\}$

Then $\forall i \in 1 .. n, (a_i \in Set\_Of\_Classes) \Lambda (d_i \in Set\_Of\_Classes)$

Let $R_R = \{<<a_1, <b_1, c_1>>, <d_1, <e_1, f_1>>>, .. ,<<a_n, <b_n,c_n>>, <d_n, <e_n, f_n>>>\}$

Then $\forall i \in 1 .. n, (a_i \in Set\_Of\_Classes) \Lambda (d_i \in Set\_Of\_Classes)$

These state that the classes must exist before they are used in a relationship, not that all classes must exist before any relationship must be defined.

This chapter defined an OORA mathematical model that included OORA components and relationships between the components. This model was used as the basis for the domain and problem model in the computer-based system called OAKS. The components of the model were also used as the basis for the development of the guidelines and rules that evaluate them. The development of the model was necessary to analyze the similarities and differences in existing OORA methods. No one existing OORA method provided the basis that could be used for the development of an automated system to conduct OORA.

Now that an OORA mathematical model has been developed that defines a set of OORA components and relationships, the next chapter reviews existing OORA methods to develop a set of guidelines and rules that can be used to evaluate the components and relationships in the model. These guidelines and rules will be refined and supplemented by

domain-specific guidelines and rules in Chapter 6, and then by guidelines and rules based on the LISP structure of the OAKS domain model in Chapter 7.

V. Acquisition and Evaluation of Components

Overview

The previous chapter defined an OORA model containing the model components and relationships. It also defined any order required on the acquiring of the components. The next step, discussed in this chapter, is an analysis of existing OORA methods. This analysis produced a set of guidelines and rules for the acquisition and evaluation of each component of the OORA model and of the entire model. These guidelines are rules are independent of any domain chosen for the domain model in the OAKS system.

This chapter discusses this analysis and the guidelines and rules resulting from the analysis. How these guidelines and rules are quantified so they can be used in a computer system is discussed in the next chapter. Also discussed in the next chapter are guidelines and rules that were developed based on knowledge of the domain used for the domain model in OAKS.

The literature surveyed provided numerous processes for both acquiring and evaluating the components. These processes were evaluated and the information combined and refined to produce the guidelines and rules discussed in this chapter. Based on an analysis of the process of acquiring problem model information, a decision was made to have OAKS develop a system by starting with a domain model as an initial template. The domain model is then modified to produce a problem model.

The elements that must be acquired, as defined by chapter 4, are:

1. Set_Of_Classes
    1.1 Class_Name
    1.2 Class_State_Space
        1.2.1 Attr_Name

1.2.2 Set of Attribute Values

1.3 Services

1.3.1 Service_Name

1.3.2 Input_Sets

1.3.3 Output_Sets

1.3.4 Preconditions

1.3.5 Postconditions

2. R_I, which is the superclass relation

3. R_W, which is the whole/part relation

4. R_R, which is other relations

The first question to be answered was what knowledge the user of OAKS should be assumed to have. This affects how the elements are acquired. It was determined that the user should have knowledge of the problem domain, the system's responsibilities, the problem scope, the application context, and any assumptions of the domain. These are reasonable assumptions given a user that is working in a particular domain.

The second question was what approach should be taken in acquiring and modeling the information. Three approaches were discussed in [MONA92]:

(1) The combinative approach. In the combinative approach, different techniques are used to model the structure, functional behavior, and dynamic behavior. Modeling techniques used include object-oriented, function-oriented and dynamic oriented. A method is defined for integrating the different approaches. This is the approach of [RUMB91] and [SHLA88]. The main problem with this approach is the difficulty encountered when trying to integrate the different views. It is difficult to determine how the different views directly relate.

(2) The adaptive approach. This approach uses existing techniques, usually from structured analysis, in a new object-oriented way, or extends these techniques to include object-orientation.

(3) The pure object-oriented approach. New techniques are used for the whole model. A translation process is not required, but a complexity-management scheme is needed to allow viewing of the data from different levels of complexity, at varying levels of detail, and from various perspectives.

The approach used in this research was the pure object-oriented approach. The combinative approach would be difficult to implement in a automated system because there would not be a human to make the mental translation from one model to another. Any modeling scheme used by OAKS would have to be tightly integrated because the same information must be used for all views and the reasoning must be done on one evolving model. The decision was already made not to use any structured analysis techniques and to assume an object-oriented perspective from the start of the requirements analysis process. This decision eliminated the adaptive approach.

The remainder of the sections in this chapter discuss each of the elements and subelements of the OORA model, how they will be acquired, and guidelines and rules for their acquisition. The formalization of these guidelines and rules for use in a computer system is postponed until the next chapter.

Set Of Classes

The first step in acquiring an OORA model is to determine the overall strategy of acquiring classes. Classes can be acquired by looking at nouns in the documents, by identifying the top-level class first and working down, by starting with the classes that interface with outside devices and working in, by identifying key abstractions and building

up the problem model from them, or by starting with a domain model and modifying this model to develop the problem model. The decision on which approach to use is partially based on how the domain information will be used in OAKS. The domain information can be used as a template for the problem model or as a comparison tool. A template is used to create a problem model by adding elements, deleting elements, and modifying the structure. The domain information can also be used for comparison with the evolving problem model in order to provide guidance and corrections.

One advantage to using the domain model as a template is that it would provide a lot of support to the user through its ability to illustrate an OORA structure. This OORA structure would serve to act as a model for a good object-oriented analysis. Modifications to the structure would be evaluated to ensure the structure remains consistent and that the changes do not violate any rules and constraints. The disadvantage of using the domain model as a template is that the user of OAKS has less flexibility in developing a new system. This is an advantage when dealing with a user who is inexperienced in object-oriented analysis. An analyst experienced with object-oriented techniques may feel constrained by this approach.

The use of the domain model as a template also assumes that the template will be valid for a reasonable period of time. If the domain model needs constant changing because the domain it represents changes rapidly, the usefulness of a system such as OAKS would be diminished. [ARAN89] states that there are stable problem domains that evolve gradually over time. Within these domains, there are communities of users that develop large numbers of software systems in the domain. These communities of users have a common vocabulary with shared semantics, and there is expertise on how to build systems in that domain. Given that one of these stable problem domains is used, it is reasonable to assume that a domain model would have a useful life. Also, further

modifications to OAKS could incorporate a learning mechanism that would update the domain model based on the problems that it was presented.

Another argument for using the domain model for a template is that a structured approach with a well-defined path, like that used by a computer system, is best suited to a well-known domain, where previous experience would provide much of the structure of the domain model. If the domain is unfamiliar, then the domain knowledge must be uncovered and refined. This process of domain discovery requires the use of a minimum number of constraints and only high-level guidance to allow for a more unconstrained and therefore creative process [WHIT90].

Using the domain model as a comparison tool requires a problem model that is less constrained as to form and content. The problem model will still have to meet the rules and constraints of the system, but the form is more flexible.

Because the user is assumed to be an analyst who is not experienced in object-oriented techniques, and because the system is computer-based, the decision was made to use the domain model as a template. Based on that decision, new classes are developed by additions to the domain model, or by changing the name or components of an existing class in the model. The process of acquisition of classes or any components in the model, is not explicitly coded in the OAKS system. Rather, the original domain model was developed by an analyst experienced in OODA using these acquisition guidelines and rules. The domain model serves as a template and provides guidance by example of the types of classes in the domain. Since classes are the most stable components in the domain model, these should be changed the least by the user of OAKS. The names of the classes may change to meet the particular problem, but the classes themselves will be relatively stable. Any acquisition guidelines and rules that would be needed by the user would be contained in a user's manual for OAKS. For example, the guidelines and rules for acquiring classes by examining the nouns in the domain would be contained in any

user's manual developed for the system, so the user can choose appropriate new classes, if needed.

For the OAKS proof-of-concept system, a user's manual was not developed. The user interface developed was sufficient to show the OAKS concepts to be sound, but the user interface is not a robust user's environment. Discussed in this chapter are the types of concepts that should be included in a user's guide for a system modeled on the OAKS concepts.

The remainder of this chapter discusses the guidelines and rules for acquiring and evaluating the components and relationships in the OORA model that is used as the basis for the domain model in OAKS.

There is much discussion in the literature on the general characteristics of classes. These characteristics are used as a basis for the guidelines and rules that evaluate a problem model. However, these general characteristics are not specific enough to be used directly by a system. For example, one general characteristic is that a class have crisp boundaries. That statement cannot be used to evaluate a class objectively, because there are no measurement criteria that can evaluate if a boundary is crisply defined or not.

Some general characteristics are as follows:

- autonomous, coherent, encapsulated, crisp boundaries [RUMB91] [BOOC91] [MEYE88].
- can be concrete or conceptual [RUMB91] [KORS90].
- have identity [RUMB91].
- tangible/visible [BOOC91].
- may be apprehended intellectually [BOOC91].
- represent the common vocabulary of the problem domain [BOOC91] [WIRF90].
- strictly controlled communication channels [MEYE88].

- designed as problem space classes, which are those needed to satisfy the requirements in an ideal environment, i.e., one large program on a fast machine [WHIT89]. This means the classes developed for requirements do not need to concern themselves with time or space requirements.

A process discussed in [SIBL89] and used in some form by others is classifying the classes as active or passive. An active class is one that can act upon other classes by sending messages. A passive class is one that accepts messages but does not send any out. Such a classification is not useful in this research. Whether the class is active or passive will be obvious from its final form. In the analysis process, there is no advantage in trying to classify the class as active or passive, and that classification may change.

Although some guidelines and rules are too general to be evaluated effectively, a set of guidelines and rules must be developed to analyze the developing model. Discussed in this chapter will be the domain-independent guidelines and rules that only require knowledge of the classes for their evaluation. Some guidelines and rules require knowledge of the domain and some require knowledge of the structure of the domain model. The guidelines and rules based on domain knowledge are discussed in Chapter 6, and those based on the OAKS domain structure are discussed in Chapter 7, after the structure is developed.

The guidelines and rules for *analyzing* classes are:

1. Name the class with a singular noun, or an adjective and a noun. The name describes a single object in the class [COYO91]. The name should not reflect the role it plays in a relationship [RUMB91].
2. Use standard terminology for class names using the common vocabulary of the domain [COYO91] [WIRF90].

3. Eliminate classes that have little or nothing to do with the problem. Keep a class if the system needs to remember anything about the objects in the class. [COYO91] [RUMB91].
4. Ensure that you are able to describe an object in the class, and some potential attributes [COYO91].
5. Ensure that the class provides some processing [COYO91]. You should be able to write a statement of purpose for the class [WIRF90].
6. Look for more than one object in a class. If there is not, look for similar classes and put the class information into those classes [COYO91].
7. Ensure that the requirements the class satisfies are domain-based requirements and not implementation constructs. Make sure the class is satisfying requirements that are needed regardless of the computer technology that will be used to build the system. Do not model windows, menus, task management, or number of processors [COYO91] [RUMB91].
8. Eliminate classes that are merely derived results. For example, you do not want a class that is a printed report consisting of existing data [COYO91].
9. Rename class names that primarily describe individual objects as attributes [RUMB91].
10. Eliminate classes that describes an operation that is applied to objects and is not manipulated in its own right, because they are not classes. A telephone call to someone is an event and not a class if you do not need to track calls [RUMB91].
11. Ensure that you can answer how objects of the class are created, copied, or destroyed [BOOC91].
12. When an adjective is used with a noun to name a class, it is probably a subclass of the noun [WIRF90].
13. Ensure that each class is not just an encapsulated subroutine [MEYE88].

14. Ensure that every object of the class has the same characteristics and is subject to the same rules [SHLA88].
15. List the criteria for object inclusion in a class. If the word "or" is used significantly, it is not a class. Also, if the criteria are just a list of objects, you don't have a class [SHLA88].

In the following sections, each component of the OORA model is discussed. First shown are the domain-independent guidelines and rules that are used for acquiring each component of the model. These would be used during the OODA in order to create the initial domain model in OAKS. These would also be placed in a user's manual so the user would have guidance on how to modify that component of the domain model, if it is needed to produce the problem model. Next shown are the guidelines and rules for evaluating each component of the model, once the component is in the model. These are used when possible in evaluating the domain and problem model in OAKS.

Superclass Relation

The superclass relation is used to create the inheritance structure for the problem model. The guidelines and rules for *acquiring* the inheritance structure are:

1. Consider each class as a potential superclass. What are its possible subclasses? For each possible subclass considered, ensure that it is in the problem domain, within the system's responsibilities, inherits the attributes and services of the superclass, and meets the requirements of a class. If there are many superclasses possible, first identify the most complex and then the simplest, and then identify the rest of the superclasses [COYO91] [RUMB91].
2. Consider each class as a potential subclass. What are its possible superclasses? For each possible superclass considered, ensure that it is in the problem domain, within

the system's responsibilities, will contain a subset of the attributes and services of the subclasses, and meets the requirements of a class [COYO91].

3. When creating subclasses, only one property should be discriminated at once [RUMB91]. This means that the differences between the subclasses on one level should be the difference in one property in the superclass. For example, if the superclass is "Aircraft", a set of subclasses at level 1 may be jet-powered aircraft and propeller-driven aircraft. From that level, the next level of subclasses at level 2 may break each of those into Air Force vs. Army aircraft. The level 1 subclasses should not be jet-powered Army aircraft, jet-powered Air Force aircraft, propeller-driven Army aircraft, and propeller-driven Air Force aircraft.
4. A superclass can be created by generalizing common aspects of existing classes into a superclass [RUMB91] [RUBI92]. These aspects include behavior, attributes, and services [WIRF90] [BULM91].
5. When several classes appear to be analogous, it is a sign that they may share a common superclass [RUMB91].

The guidelines and rules for *evaluating* the inheritance structure are:

1. The distinctions between subclasses must be important within the problem domain. Specialize around those important distinctions. For example, it may be important to distinguish between dogs and cats and not between male and female pets [COYO91].
2. Some set of attributes and services must be common to all the subclasses of a superclass [COYO91]. Subclasses should support all the responsibilities of their superclasses [WIRF90].
3. If the only distinction between two subclasses is the value of one attribute, then just use the superclass with different values of one attribute. For example, if the only difference between two types of pets is whether they are male or female, just use a "sex" attribute in the superclass and remove the subclasses [COYO91].

4. If multiple inheritance is used, the subclass does not have to add attributes or services to be a good subclass [COYO91].
5. The inheritance structure should reflect naturally occurring structure in the domain. Do not use inheritance just to extract out a common attribute [COYO91].
6. Do not nest subclasses too deeply. Look suspiciously at those that are over three levels deep [RUMB91].
7. Services may not change the Input_Set and Output_Sets, but they may change their behavior [RUMB91]. If a subclass redefines a service inherited from a superclass, it may redefine the behavior of the service, but it may not redefine the form of the Input_Set and the Output_Set.
8. Factor a common responsibility as high as possible in the inheritance hierarchy [RUMB91].
9. There should be at least two subclasses per superclass [RUMB91].
10. If there is trouble naming a superclass, there is probably a problem. Try another superclass [WIRF90].

Whole/Part Structure

The guidelines and rules for *acquiring* the whole/part structure are:

1. Look for assembly parts, container contents, and collection members [COYO91].
2. Consider each object as a whole. For each potential part, ensure that it is in the problem domain, within the system's responsibilities, captures more than just status value, and provides a useful abstraction [COYO91].
3. Consider each object as a part. For each potential whole, ensure that it is in the problem domain, within the system's responsibilities, captures more than just status value, and provides a useful abstraction [COYO91].

4. Ask if you would use the phase "part of" in an association between two classes [RUMB91].

The guidelines and rules for *evaluating* the whole/part structure are:

1. If a part does not capture more than just status value, include an attribute for that value in the whole and eliminate the part [COYO91].
2. There can be some operations on the whole that are applied to its parts, but never from the parts to the whole [RUMB91].

Class State Space

Guidelines and rules for *acquiring* the class state space are:

1. Ask how each class is described in general [COYO91].
2. Ask how each class is described in this problem domain [COYO91].
3. Ask how each class is described in the context of the system's responsibilities [COYO91].
4. Ask what each class needs to know to function [COYO91].
5. Ask what state information needs to be remembered over time [COYO91].
6. Ask what states can each class be in. These states are represented by attribute values [COYO91].
7. Put the attribute in the uppermost class in an inheritance structure where it remains applicable to all subclasses [COYO91].
8. Use the standard vocabulary of the problem domain to name attributes [COYO91].
9. Attributes are described by their name, type and default value [COYO91] [RUMB91].
10. Attributes usually correspond to nouns followed by possessive phrases, such as "The color of the car" [RUMB91].

11. Define all the characteristics that each object of the class possesses and what information is needed to know if an object is an instance of a particular class [SHLA88].

Guidelines and rules for *evaluating* the class state space are:

1. Each attribute should represent an atomic concept in the form of either a single value or a tightly related group of values [COYO91] [SHLA88].
2. The attributes should apply to every object in the class. If not, create another set of classes using inheritance [COYO91].
3. Attributes should not be derived results, such as Age when you know the date of birth [COYO91].
4. Data redundancy is acceptable during the analysis phase [COYO91].
5. If attributes are repeated in other classes, there may be additional classes required in the inheritance structure [COYO91].
6. Do not use internal identifiers as attributes. The object IDs are implicit in that they are assumed for every object [COYO91] [RUMB91]. Internal identifiers have no meaning in the problem domain.
7. An attribute should be a class if the independent existence of an entity is important rather than just its value [RUMB91].
8. If an attribute describes an internal state that is invisible outside the object, eliminate it [RUMB91].
9. An attribute that is completely different from and unrelated to other attributes may indicate the class should be broken into two classes [RUMB91].
10. Each attribute should be independent of the other attributes in a class [SHLA88].
11. Each attribute should take on only one value at a time [SHLA88].
12. There must be a value for every attribute [SHLA88]. The possible value for an attribute should not be N/A.

Other Relationships

Guidelines and Rules for *acquiring* other relationships are:

1. Look for a tie between objects that is used to satisfy a responsibility of the system [COYO91].
2. Look for dependencies between classes [RUMB91].
3. Relationships often correspond to verb or verb phrases. These include physical location (next to), directed actions (drives), communications (talks to), ownership (has), or satisfaction of some condition (works for, married to, manages).
4. In a one-to-one relationship, take an identifier in one class and make it an attribute in another. For example, each state has a governor. Put the attribute State_Name in class Governors [SHLA88].
5. In one-to-many relationships, take an identifier from the "one" class and make it an attribute in the "many". For example, place Owner's name as an attribute in the Dog class [SHLA88].

Guidelines and rules for *evaluating* other relationships are:

1. When there is a many-to-many relationship either between objects of different classes or objects for a single class, ask what attributes describe the connection. Then make a class between the two connected classes that contains those attributes [COYO91] [BULM91] [SHLA88].
2. Do not add a relationship if the mapping between two objects can be made through other relationship connections [COYO91] [RUMB91].
3. Challenge one-to-one relationships. Often the object on either end is optional or multiplicity is needed [RUMB91].

4. Eliminate relationships that are outside the problem domain or deal with implementation [RUMB91].
5. A relationship should not describe a transient event, but a permanent relationship [RUMB91].

The uncovering of relationships is not as obvious as finding the other components of the model discussed so far. This is because the relationships that are needed are highly problem dependent. This implies that possible relationships should be encoded in the domain model and used as guidance for the possible creation of new relationships. The issue will be whether these new relationships will be the reused relationship names used on a different pair of classes, or whether new relationship names will have to be added to the model. The more difficult task for the user of OAKS will be the adding of new relationship names to the model. The user will also have to define the multiplicity of all new and modified relationships.

Services

Service information will be the most difficult to obtain. This is because the possible object states as well as the pre- and post-conditions will have to be obtained in some form. The form that it will take and how it might be acquired will be discussed in the next chapter. In this section only the basic information that must be acquired and guidelines and rules for evaluation once the acquisition is complete are discussed.

Guidelines and rules for *acquiring* the services are:

1. The process requires the steps of identifying the possible states, identifying the service names, identifying what services of other classes are needed for the service to perform its function, and then identifying the pre- and post-conditions in some form. To identify the states, the potential values for the attributes are examined to determine

whether there is different behavior for those potential values. When there is different behavior, there exists some state. There are two type of services: algorithmically-simple and algorithmically-complex. The simple services consist of those that create an object, get or set an attribute value, or delete an object. The complex services calculate or monitor.

In order to identify the message connections, ask:

- what other object does it need services from?
- what other objects need one of its services? [WIRF90]

Last, the services are specified. The pre-condition will show the states in which the service is valid and the service arguments. The post-condition shows the results of the service and what other classes are needed (message passing) [COYO91].

2. Identify the input and output values of the service. Show how input values are computed from output values. Specify pre- and post-conditions. Specify optimization criteria [RUMB91].
3. A class of free programs may be created. These are services that are useful to more than one class. It reduces the coupling between classes [BOOC91].
4. Look at the verbs in the requirements specification for possible service names [WIRF90].
5. Examine how the system will be invoked. Go through a variety of scenarios using all system capabilities [WIRF90].
6. Look at each class and ask what responsibility it was created to satisfy. What responsibilities are required for managing its attributes? Compare and contrast the roles of various classes [WIRF90].
7. If more than one class must maintain the same information, then either create a new class that has the common information, assign the responsibility to one class if it is the

primary behavior for one of the classes, or collapse the different classes into one class [WIRF90].

8. To identify message passing, ask if each class is capable of fulfilling its responsibilities itself. If not, what else does it need and what classes provide this information [WIRF90].
9. To design the service interface:

   - Define the most general message; one that allows clients to supply all possible required parameters.

   - Provide default values for any parameters where it seems reasonable to do so.

   - Analyze how clients use the messages. Define messages that allow clients to specify only some of the parameters while providing default values for others [WIRF90].

   - Design services with a single purpose [WINB90].

Guidelines and rules for *evaluating* services are:

1. Look at possible reusability of the service. Ask if the service would be useful in more than one context. Try to make the services as reusable as possible [BOOC91].
2. Look at the complexity of the service and ask how difficult it would be to implement. You may have to break the service into two services [BOOC91] [WINB90].
3. Ask how applicable the service is to the class in which it is placed [BOOC91].
4. Make sure that the implementation of a service does not depend on the internal details of another class [BOOC91].
5. Each service should send messages to a limited set of classes. This creates loosely coupled classes [BOOC91].
6. Services should be named with active verb phrases [BOOC91].
7. Make sure all known system actions are accounted for through service actions [WIRF90].

8. The intelligence of a system should be evenly distributed. Intelligence is measured by how much a class knows or can do, and how many objects it can affect [WIRF90].
9. Keep services with related information. If a class has attributes, then the services that manipulate those attributes should be in the same class. If a service requires certain information, then that information should be in its class [WIRF90].
10. If a class has no message connections with other classes, it should be discarded. Be sure necessary message connections have not been overlooked [WIRF90].
11. Services should not have to check the class of an object [WINB90].
12. A service should not have more than six arguments. Reduce the number of arguments by breaking the service into several [WINB90].
13. Keep the amount of work a service does to a minimum. Smaller services can be selectively inherited, refined, or overridden [WINB90].
14. Identify common services and put them in a superclass [WINB90].
15. Eliminate from a superclass those services that are frequently overridden rather than inherited by its subclasses [WINB90].
16. Services should apply to all the objects in a class. If not, the inheritance structure needs to be modified [COYO91].
17. If a class has too many services, break it into multiple classes [WHIT89].

Whole Model

The whole model is acquired through the acquisition of the components of the model. This is done as the OAKS domain model is created, prior to its use in OAKS as a template. However, the whole model must be evaluated both when it is created and after each change is made by the OAKS user.

The guidelines and rules for *evaluating* the whole model are:

1. One attribute in a class is suspicious. It is likely that attribute should be included in other classes and that class removed [COYO91].
2. A class should have services other than just create and destroy [COYO91].
3. Weakly coupled classes are desirable, but there is a tension between weak coupling and inheritance. You want a minimum of message passing [BOOC91] [CHID91].
4. A class should be highly cohesive. Preferably, functional cohesion is used where all elements of the class work together to provide some well-bounded behavior [BOOC91]. Also, you want the union of the set of instance variables used by all the services of a class to be as large as possible [CHID91].
5. All services of a class should be primitive [BOOC91].
6. The complexity of a class is measured by the total number of attributes and services. The complexity of a class should be kept low [BOOC91].
7. Eliminate redundant classes. These are two or more classes that encapsulate the same information [RUMB91] [BULM91].
8. Redefine classes that have ill-defined boundaries or are too large in scope [RUMB91].
9. Make sure that every functional requirement is met by the classes [BAIL89] [WIRF90] [RUBI92].
10. If a class does not have a rich set of services, then it may be better to put its attributes and services in other classes [WALT78].

This chapter evaluated existing OORA methods and, based on these methods, defined a set of guidelines and rules for the acquisition and evaluation of components of the OORA mathematical model. The next chapter evaluates these guidelines and rules and determines which, and in what form, can be used in the computer-based OORA system OAKS. The next chapter also defines guidelines and rules that are based on the specific domain chosen for an OAKS domain model.

VI. Application of Evaluation Guidelines and Rules

Overview

The previous chapter defined a set of domain-independent guidelines and rules that could be applied to evaluate an OORA model of a system. The problem is that many of the guidelines and rules are subjective and must be defined objectively before they can be used by a computer-based system such as OAKS. This chapter examines each of the evaluation guidelines and rules of the previous chapter and determines how they can be applied in the OAKS system. This chapter also defines guidelines and rules that are defined based on the domain currently in use by OAKS.

Also defined in the previous chapter were a set of guidelines and rules on how to acquire components of the OORA model. These guidelines and rules for acquiring an OORA model component are not used in checking the domain and evolving problem model in OAKS. These are embodied in how the initial domain model is constructed and should be placed in any user's guide developed for an OAKS-based system. An example of this type of guideline and rule is that the class names should use the common vocabulary of the domain. The evaluation guidelines and rules can be checked when the domain model is created and as the problem model is evolving. For example, classes unconnected with other classes are identified.

The last set of guidelines and rules is discussed in Chapter 7. These are based on the code structure of the domain model and problem model in OAKS. These are applied to both the domain model and the evolving problem model.

In the following sections, the evaluation guidelines and rules from the previous chapter are annotated by "GR" and the number that was used in the previous chapter. Their use in the OAKS model is annotated by "USE" before their number.

Evaluating Classes

Most of the guidelines and rules defined for evaluating classes are embodied in the domain model and in the process OAKS uses for acquiring new classes based on that model. For example, the existing classes in the domain model will be named appropriately with descriptions of properties and purpose attached to them. However, some of the guidelines and rules will need to be applied when a new class is created.

GR1. The class name is a singular noun, or an adjective and a noun. It describes a single object in the class [COYO91]. The name should not reflect the role it plays in a relationship [RUMB91].

USE1. This is checked crudely by checking for "s" endings on the name. Classes using an adjective and a noun should have an underscore between the words. Eventually, a system such as OAKS will require a parser for the user input that can analyze whether the noun is singular or plural, and to ensure the leading word is an adjective.

GR2. Use standard terminology for class names using the common vocabulary of the domain [COYO91] [WIRF90].

USE2. In a user's guide, the user would be instructed to name new classes using standard terminology. Also the user's guide should have the user ensure the new class is not just a new name of an existing class in the domain model.

GR3,8,9,10,13. It is a class if the system needs to remember anything about the objects in the class. Eliminate classes that have little or nothing to do with the problem [COYO91] [RUMB91]. A class should not merely be derived results. For example, you do not want a class that is a printer report from existing data [COYO91]. Class names that primarily describe individual objects should be renamed as attributes [RUMB91]. A name that describes an operation that is applied to objects and is not manipulated in its

own right is not a class. A telephone call to someone is an event if you do not need to track calls [RUMB91]. A class should not just be an encapsulated subroutine [MEYE88].

USE3,8,9,10,13. The domain model provides guidance to the user on what are proper classes for the domain through the use of the existing classes, the inheritance structure, and the whole/part relations. The user is not intended to be an object-oriented expert, so the domain model is used as an example of where to look for classes and how they fit together. Also, if there are classes that have no connections to other classes when the problem model is complete, those classes are either not needed for the problem or there are relations between classes not yet defined. Any classes not connected to other classes are brought to the user's attention for resolution.

GR4,5,11. You are able to describe an object in the class, and some potential attributes [COYO91]. The class needs to provide some processing [COYO91]. You should be able to write a statement of purpose for the class [WIRF90]. You should be able to answer how objects of the class are created, copied, or destroyed [BOOC91].

USE4,5,11. The user is asked for an English description of the class that included its general properties, its purpose, and the processing it needed to do. If a user can provide this information, then it is probably a good class. The English description is not analyzed in OAKS, because of the absence of a parser, but it is stored with the new classes. Existing classes in the domain model carry this information also. It is not reasonable to ask a user how objects are created, copied, or destroyed. This is based more on how the system is designed and on the implementation language used than with requirements.

GR6. There usually should be more than one object in a class. If not, look for similar classes and put the class information into those classes [COYO91].

USE6. The user's guide should advise the user to create a potential set of objects for any new class. If there are no possible objects, or just one, the class may have to be redesigned.

GR7. The requirements the class satisfies should be domain-based requirements and not implementation constructs. Make sure the class is satisfying requirements that are needed regardless of the computer technology that will be used to build the system. Do not model windows, menus, task management, or number of processors [COYO91] [RUMB91].

USE7. The user should be cautioned through a user's guide about creating a class that is implementation-dependent. This requires providing the user with examples of implementation-dependent classes in that domain.

GR12. When an adjective is used with a noun to name a class, it is probably a subclass of the noun [WIRF90].

USE12. This is used to analyze the inheritance relation. The user is asked to place the underscore character between words composing a class name. Matches are made on the words making up the name.

GR14. Every object of the class must have the same characteristics and be subject to the same rules [SHLA88].

USE14. A user's guide should instruct the user that all the objects of a class should have the same characteristics.

GR15. List the criteria for object inclusion in a class. If the word "or" is used significantly, it is not a class. Also, if the criteria are just a list of objects, you don't have a class [SHLA88].

USE15. This information should be placed in the user's guide.

Evaluating the Inheritance Relation

GR1. The distinctions between subclasses must be important within the problem domain. Specialize around those important distinctions. For example, it may be important to distinguish between dogs and cats and not between male and female pets [COYO91].

USE1. The domain knowledge in OAKS is in the form of the domain model. When new classes are acquired, they are acquired with the purpose of gathering information that is needed for the problem model. The domain model acts as a template, or model, for how the classes and the inheritance structure should be organized for that domain.

GR2,7. Some set of attributes and services must be common to all the subclasses of a superclass [COYO91]. Subclasses should support all the responsibilities of their superclasses [WIRF90]. Services may not change the Input_Set and Output_Sets, but they may change their behavior [RUMB91].

USE2,7. The inheritance structure in the domain model requires that subclasses inherit all attributes and services from their superclasses, although the services may be implemented differently. The requirement is that the service interface is the same for the subclasses of a superclass. If the user creates a new class, it must obey all these rules also.

OAKS could be flexible enough to allow the superclass to be redefined so that the subclass would fit under that superclass. The problem with this approach is that this may invalidate large segments of the domain model. If the superclass is changed, all subclasses under it must be redefined. This is a large task for a user not familiar with object-oriented techniques. This would create the possibility for the user to do major damage to the domain model. If the user is considered to be an object-oriented expert, the system could be more open and allow any changes desired. However, given the assumed expertise of the user, OAKS does not allow changes to classes that are superclasses of other classes in the domain model. OAKS permits changes to classes that are not superclasses as long as those changes do not violate the constraint that the class must inherit all attributes and

services from its superclass. For example, assume class Aircraft is a leaf of the domain model inheritance tree and the user wants to create a class Helicopters that is a subclass of Aircraft. Further assume that Aircraft has an attribute Wing_Span that is NOT an attribute inherited from its superclasses, and further assume that the Helicopter class has no use for this attribute, but can inherit all other attributes and services of the Aircraft class. The user of OAKS could remove the Wing_Span attribute from Aircraft and create a class Fixed_Wing_Aircraft as a subclass of Aircraft that contains the Wing_Span attribute. The helicopter class can now be made a subclass of the Aircraft class.

If the user defines a new class that does not fit under any of the existing classes in the model, the new class is created by the user as an independent class.

GR3. If the only distinction between two subclasses is the value of one attribute, then just use the superclass with different values of one attribute. For example, if the only difference between two types of pets is whether they are male or female, just use a "sex" attribute in the superclass and remove the subclasses [COYO91].

USE3. The analyst that creates the domain model uses OAKS to determine if there are a large number of attributes of any two classes that are similar in structure. If there are, these are identified by OAKS so the analyst can determine if the two classes are related and if the model needs to be changed.

GR4. If multiple inheritance is used, the subclass does not have to add attributes or services to be a good subclass [COYO91].

USE4. If a class is added using single inheritance, OAKS ensures that the new class has at least one new attribute and/or service. But if a class is added using multiple inheritance, OAKS does not use this requirement.

GR5. The inheritance structure should reflect naturally occurring structure in the domain. Do not use inheritance just to extract out a common attribute [COYO91].

USE5. The inheritance structure in the domain model follows the guideline.

GR6. Do not nest subclasses too deep. Look suspiciously at those that are over three levels deep [RUMB91].

USE6. The inheritance structure of the domain model minimizes the nesting of the subclasses. Also, the inheritance structure of the domain and problem model is analyzed and too deep a nesting level is flagged to the developer of the OAKS domain model and the user for possible model changes.

GR8. Factor a common responsibility as high as possible [RUMB91].

USE8. This guideline is used in the development of the domain model. Also, the model is evaluated to determine if there are a large number of services in any two classes that are similar in structure. If there are, this information is provided to the developer of the domain model and the user of OAKS.

GR9. There should be at least two subclasses per superclass [RUMB91].

USE9. This guideline is used in the development of the domain model and also in any structure that the user creates. If a class only has one subclass, that is flagged and brought to the user's attention for possible model changes.

GR10. If there is trouble naming a superclass, there is probably a problem. Try another superclass [WIRF90].

USE10. The user must identify names for each new class. There is no method for determining if the user had difficulty in determining a name or not. The users guide should provide the guidance to the user in naming classes.

Evaluating the Whole/Part Relation

GR1,2. If a part does not capture more than just status value. include an attribute for that value in the whole and eliminate the part [COYO91]. There can be some operations

on the whole that are applied to its parts, but never from the parts to the whole [RUMB91].

USE1,2. The whole/part structure of the domain model follows this guideline.

Evaluating the Class State Space

GR1,3,4,6,7,8,9,10. Each attribute should represent an atomic concept in the form of either a single value or a tightly related group of values [COYO91] [SHLA88]. Attributes should not be derived results, such as Age when you know the date of birth [COYO91]. Data redundancy is acceptable during the analysis phase [COYO91]. Do not use internal identifiers as attributes. The object IDs are implicit in that they are assumed for every object [COYO91] [RUMB91]. Internal identifiers have no meaning in the problem domain. An attribute should be a class if the independent existence of an entity is important rather than just its value [RUMB91]. If an attribute describes an internal state that is invisible outside the object, eliminate it [RUMB91]. An attribute that is completely different from and unrelated to other attributes may indicate the class should be broken into two classes [RUMB91]. Each attribute should be independent of the other attributes in a class [SHLA88].

USE1,3,4,6,7,8,9,10. These guidelines and rules are used by the researcher when developing the attributes used in the OAKS domain model. The users guide should provide information on the proper selection of attributes. Attributes are more likely to change than classes from problem to problem in a domain, so the attribute structure is more likely to change than the class or inheritance structures.

GR2,11,12. The attributes should apply to every object in the class. If not, create another set of classes using inheritance [COYO91]. Each attribute should take on only

one value at a time [SHLA88]. There must be a value for every attribute [SHLA88]. The possible value for an attribute should not be N/A.

USE2,11,12. The OAKS domain model adheres to the guideline that the attributes apply to every object in the class. The domain model also forces a legal set of values for each attribute, which ensures the attribute must have a value, but only one value at any point of time.

GR5. If attributes are repeated in other classes, there may be additional classes required in the inheritance structure [COYO91].

USE5. This guideline is used internally by OAKS to evaluate the initial domain model. Also, attributes in new classes that are used in other classes point to possible relations between them. OAKS brings these similarities to the attention of the developer of the OAKS domain model for possible relations or for possibly combining the new class with another.

Evaluating Other Relationships

GR1,2,3,4,5. When there is a many-to-many relationship either between objects of different classes or objects for a single class, ask what attributes describe the connection. Then make a class between the two connected classes that contains those attributes [COYO91] [BULM91] [SHLA88]. Do not add a relationship if the mapping between two objects can be made through other relationship connections [COYO91] [RUMB91]. Challenge one-to-one relationships. Often the object on either end is optional or multiplicity is needed [RUMB91]. Eliminate relationships that are outside the problem domain or deal with implementation [RUMB91]. A relationship should not describe a transient event, but a permanent relationship [RUMB91].

USE1,2,3,4,5. Other relationships are highly domain dependent. Where inheritance and whole/part relationships are used in almost all domains, the other relationships used, if any, are based on the domain. Each domain carries with it a set of other relations that are normally used in that domain that are represented in the domain model. New relationships are added by the user of OAKS as needed. Also, OAKS allowes the modification of the existing other relationships, to include changing any of the components and deleting an entire relationship. The user's guide should provide guidance on how to identify other relationships.

Evaluating Services

In analyzing examples of pre- and post-conditions of services for the domain model, several points became clear. First of all, the only information needed for OAKS to analyze the services is the identification of services of other classes that are used. This provides information on the coupling between classes and the execution flow starting from a given service. Second, acquiring the algorithms from a user would be difficult, and would be better done using a tool specifically designed for that purpose and then placing the results in the OAKS model. The algorithms can take a number of forms, from pseudo-English to a more structured program design language to a higher-order language like Ada, depending on the experience of the user. Third, it is possible to acquire the preconditions from the user by requesting information on the required values (if any) of the attributes prior to service execution. Fourth, it is also possible to ask the user what attributes (if any) change as a result of the service execution.

Based on these conclusions, the following information is acquired from the user on services:

(1) The name of the service.

(2) The input parameters (if any). This includes the name of the parameter and its type. The type is the legal set of values the parameter can assume.

(3) The output parameters (if any). This includes the name of the parameter and its type.

(4) The preconditions. This takes the form of the required values (if any) of the attributes of the class. It is assumed that the input parameters are of their respective type so that the checking of the type of the input parameters does not have to be explicitly shown. For example, if the input parameter to a service called "Change_Age" is the new age, which is an integer between 0 and 100, it is assumed any input parameter value will be an integer in that range.

(5) The postconditions containes two parts. One part is information on the services of other classes required for this service to perform its function. This takes the form of the class name and the service name. The second part is information on the new state of the class upon completion of the service function. This is the changes in attribute values (if any) of the class.

This information is acquired by direct questions for the name of the service, the input parameters, the output parameters, the precondition and the postcondition, or through the use of templates for certain types of services. For services created without templates, checking is done on some of this information to ensure it is consistent with the model. For example, any attribute name used in defining the new state of the class in the postcondition had to exist in the class. The service name could not be the same as any other service name in that class. Also, the messages must be from existing classes and services within those classes. Other checks that are made on new services are described in more detail in Chapter 7.

Templates for services were created that greatly simplify the creation of a new service. Templates for services that change attribute values and return attribute values are provided in OAKS. These templates automatically fill in the values for the description, input set, output set, preconditions and postconditions given information on the type of template and the attribute the template operates on.

GR1. Look at possible reusability of the service. Ask if the service would be useful in more than one context. Try to make the services as reusable as possible [BOOC91].

USE1. This should be one of the primary considerations when the domain model is developed. It is desirable to reuse as many of the services in the domain model as possible in the problem models of that domain. Therefore, the user of OAKS is not forced to create new services or make extensive changes to existing services.

GR2,3,4,5,6,8,9,11,12,13,15,16,17. Look at the complexity of the service and ask how difficult it would be to implement. You may have to break the service into two services [BOOC91] [WINB90]. Ask how applicable the service is to the class in which it is placed [BOOC91]. Make sure that the implementation of a service does not depend on the internal details of another class [BOOC91]. Each service should send messages to a limited set of classes. This creates loosely coupled classes [BOOC91]. The intelligence of a system should be evenly distributed. Intelligence is measured by how much a class knows or can do, and how many objects it can affect [WIRF90]. Keep services with related information. If a class has attributes, then the services that manipulate those attributes should be in the same class. If a service requires certain information, then that information should be in its class [WIRF90]. Services should not have to check the class of an object [WINB90]. A service should not have more than six arguments. Reduce the number of arguments by breaking the service into several [WINB90]. Keep the services small. Smaller services can be selectively inherited, refined, or overridden [WINB90]. Services should be named with active verb phrases [BOOC91]. Eliminate from a

superclass those services that are frequently overridden rather than inherited by its subclasses [WINB90]. Services should apply to all the objects in a class. If not, the inheritance structure needs to be modified [COYO91]. If a class has too many services, break it into multiple classes [WHIT89].

USE2,3,4,5,6,8,9,11,12,13,15,16,17. The use of the service templates provides a structure for these services that followed the guidelines for services. The existing services in the domain model attempt to capture atomic concepts so that they can be more easily reused and understood by the user. The users guide should provide guidance for creation of a service if the user needs to create a service without the use of a service template.

GR7. Make sure all known system actions are accounted for through service actions [WIRF90].

USE7. OAKS asks the user for a list of all the services of other classes that are needed for each service to perform its function. This process helps in identifying services that are missing from classes. OAKS also makes it possible to follow the flow of a system action through the problem model and present that flow to the user for validation. This requiress the user to provide an initial state and a stimulus to the system. Shown are the classes that are affected and the services used and the order in which the services are invoked, as well as state changes.

GR10. If a class has no message connections with other classes, it should be discarded. Be sure necessary message connections have not been overlooked [WIRF90].

USE10. OAKS examines the problem model to identify classes that are unconnected with other classes through message connections. These classes are brought to the user's attention. The user must then determine if the class is needed, and if the class is needed, what message connections have not yet been modeled.

GR14. Identify common services and put them in a superclass [WINB90].

USE14. The domain model is created using this guideline. Also, OAKS provides a list of classes that may be related by examining if a majority of the services of any two classes are similar in structure.

Evaluating the Whole Model

GR1. One attribute in a class is suspicious. It is likely that attribute should be included in other classes and that class removed [COYO91].

USE1. The completed problem model is evaluated and all classes with just one attribute identified and brought to the user's attention. The user determines if this is acceptable or not.

GR2,10. A class should have services other than just create and destroy [COYO91]. If a class does not have a rich set of services, then it may be better to put its attributes and services in other classes [WALT78].

USE2,10. The completed problem model is evaluated for classes with no services since the services of create and destroy are not explicitly defined but are assumed to be part of every class. The user determines if the class is needed in the problem model. If it is needed, there may be other services are not identified. If the class is eliminated, the attributes of that class, if they are needed for the problem, are relocated in other classes, or a new class or classes defined that containes the attributes but with their own set of services.

GR3. Coupling. Weakly coupled classes are desirable, but there is a tension between weak coupling and inheritance. You want a minimum of message passing [BOOC91] [CHID91].

USE3. The domain model attemptes to keep the coupling between classes low by the design of the classes themselves and the services.

GR4. A class should be highly cohesive. Preferably, functional cohesion is used where all elements of the class work together to provide some well-bounded behavior [BOOC91]. Also, you want the union of the set of instance variables used by all the services of a class to be as large as possible [CHID90].

USE4. The domain model containes classes that are cohesive, and preferably, functionally cohesive.

GR5. All services of a class should be primitive [BOOC91].

USE5. See USE2,3,4,5,6,8,9,11,12,13,15,16,17 of Services.

GR6. The complexity of a class is measured by the total number of attributes and services. The complexity of a class should be kept low [BOOC91].

USE6. This guideline is used in the development of the domain model. OAKS provides a procedure that analyzes the complexity of classes and brings overly complex classes to the domain developer's attention.

GR7. *Eliminate redundant classes. This is two classes that encapsulate the same* information [RUMB91] [BULM91].

USE7. The domain model is examined by OAKS for classes that contain the same attributes and services, or share a majority of the attributes and services. It is possible that these classes could be combined. These classes are brought to the domain developer's attention.

GR8. Redefine classes that have ill-defined boundaries or are too large in scope [RUMB91].

USE8. This guideline is used in developing the domain model.

GR9. Make sure that every functional requirement is met by the classes [BAIL89] [WIRF90] [RUBI92].

GR9. See USE7 of Services.

Domain-Dependent Guidelines and Rules

The domain-dependent guidelines and rules are based on the requirements on the type of information required in the domain model for a particular domain. The domain-dependent guidelines and rules will change for each domain of interest implemented for OAKS. The domain chosen for this research is that of a system that manages the scheduling of maintenance and flights for an Air Force maintenance squadron. The domain model used as a basis for this research is shown in Appendix A. This domain model provided the information for the implemented domain model within OAKS.

The domain-dependent guidelines and rules contain information on those classes and any of its attributes and services that are necessary to the completed model and therefore cannot be deleted from the model. The names and components of these classes, attributes and services can change, therefore allowing the user to adapt these structures to the problem being solved. The classes in the model are the most likely not to change from one problem in the domain to another, and the most likely to be required in a domain. For example, if the domain is an aircraft maintenance squadron, the "aircraft" class would be required in all problems in the domain. There would also be an attribute that would represent some identification of the aircraft, such as the tail number. The class and the attribute would be required, even though the name, "tail-number" could change if that terminology is not used in the problem.

Relationships are more likely to change than classes, and therefore are not included in the domain-dependent guidelines and rules. Also, inheritance is not included because classes that are parents are not allowed to be deleted from the model. These restrictions could be added if deemed necessary for a particular domain. For purposes of illustration of the concept of domain-dependent guidelines and rules, the most likely constant structures were chosen for implementation.

If a new class is added that is not in the domain model, OAKS cannot currently apply any domain-dependent guidelines and rules.

The process of determining which classes, attributes and services are necessary requires an extensive domain analysis which is beyond the scope of this research. Therefore, certain classes, attributes and services of the domain model were chosen to be used as an example of how they would be used in the OAKS system.

The groundwork has now been laid for an automated OORA system. Starting with the OORA mathematical model, which defined components, relationships and ordering, guidelines and rules have been defined on the acquisition and evaluation of those components. The guidelines and rules are based on OORA concepts independent of the domain in which they operate, and guidelines and rules have been defined based on the domain of interest. The next step, discussed in the next chapter, is the development of a code structure for the domain and problem model in OAKS based on the OORA mathematical model, the methods for analyzing that structure using the guidelines and rules defined in this chapter, and the development of the user interface. Also included in the next chapter is the development of guidelines and rules based on the code structure for the domain and problem model in OAKS.

VII. Prototyping, Testing and Analysis

Overview

The purpose of the research was to investigate the feasibility of a computer-based system assisting in the OORA process. This required the development of a proof-of-concept computer-based system, which was called OAKS. It would not be sufficient to define the OORA model and the guidelines and rules without creating a system that implements the defined process. The proper selection of code structures, code organization, and techniques for evaluating the model in code is crucial to the achievement of the goals of this research. This chapter discusses the development of the OAKS software.

The OAKS system contains a domain model that is modified by the user of OAKS to produce a specification for a particular problem in that domain, called the problem model. The domain model is created by an analyst after conducting an OODA. The OODA process would follow the guidelines and rules outlined in the previous chapters. The components of the domain model are those of the mathematical OORA model developed in chapter 4. The analyst should create the domain model and examine the results using the code that analyzes the model based on the guidelines and rules. This would be done prior to the domain model being modified by a user. As changes are made by the user, the analysis code continues to identify any deviations from the desired final model.

The proper selection of the code structures to implement the OORA mathematical model components, the ordering of the component acquisition, and the guidelines and rules were critical to the successful implementation of OAKS. Selection of certain code structures, such as frames, would make the task of creating a pure object-oriented system very difficult. The code needed to represent the object-oriented concepts and structures

as naturally as possible. The code structures must be in a form so any changes made to the domain model can be checked for their consistency and completeness using the guidelines and rules. It was also important to organize the code so new domains can be implemented with a minimum of impact on the code that is non-domain specific.

Discussed in this chapter is the structure of the domain model, the additional guidelines and rules used to analyze both the domain and problem models based on the structure of the models in OAKS, the implementation of the guidelines and rules defined in chapter 6, the permissible modifications to the domain model used to produce the problem model, and the user interface. The guidelines and rules used to analyze the problem and domain models are those defined in chapter 6 plus those introduced in this chapter that are based on the structure of the domain model. These guidelines and rules together insure that the model remains consistent and complete in accordance with the OORA mathematical model and the desired OORA process.

LISP was chosen as the implementation language for OAKS because of its ease of use as a prototyping language and its flexibility. LISP provides the structures and environment necessary for the OAKS development.

OAKS contains five primary code structures. The first, called the domain model, contains the domain model that is modified to create the problem model and any domain-dependent guidelines and rules that are used to evaluate the user's evolving problem model. This is the only code structure that must be reimplemented for each domain. The remaining four code structures remain constant across all domains.

The second code structure, called the domain-independent guidelines and rules, contains domain-independent guidelines and rules that are used to evaluate the model to ensure it meets object-oriented requirements analysis, general requirements analysis objectives, and requirements based on the structure of the domain model. These rules evaluate any new domain model in OAKS and the user's evolving model.

The third code structure is the model modification code. This code contains the functions that allow the modifications to the domain model and the evolving problem model. The model modification functions use the code implementing the domain-independent and domain-dependent guidelines and rules to ensure that changes meet all system requirements.

The fourth code structure, the evolving model, is the user's evolving model. This model will start as a copy of the domain model. It is modified by the user using the OAKS system and, once modified, stored separately from the domain model.

The fifth code structure, the user interface, handles all communication between OAKS and the user. Figure 7-1 shows the general structure of OAKS.

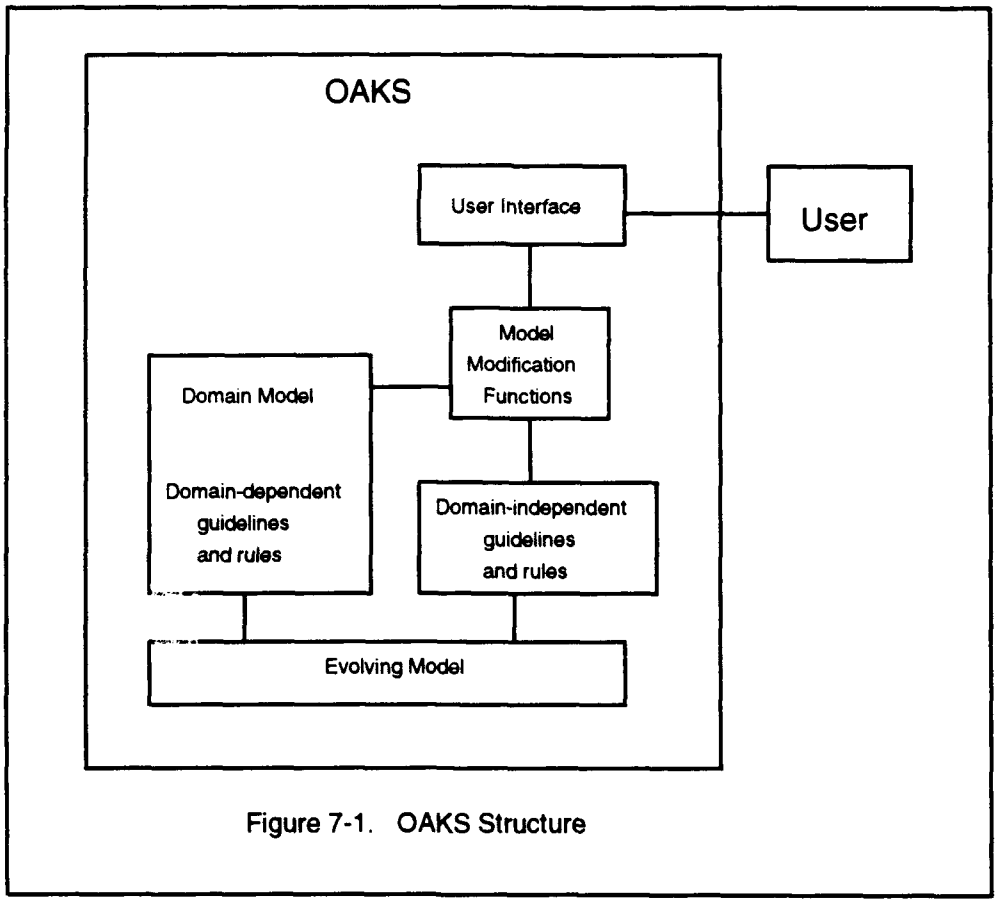


Figure 7-1. OAKS Structure

The following sections will discuss and analyze the five code structures. The implementation and analysis of the resulting structure of the domain model is covered first. Next, the implementation of the domain-independent guidelines and rules is discussed and analyzed. The domain-dependent guidelines and rules are discussed and analyzed next. The evolving model is treated in the context of the code structure that allows modifications to the domain model. Therefore, these two code structures are discussed and analyzed together. Although the user interface has nothing to do with the functionality provided by OAKS, it is important because it establishes how the user communicates with OAKS. This chapter concludes by looking at the user interface developed to complement this research.

Domain Model Code Structure

The OAKS domain model contains the information for a domain with the following components. These components were defined as in the OORA mathematical model. This list contains the components themselves, but does not show multiple occurrences of a component. For example, there are a number of classes in Set_Of_Classes.

1. Set_Of_Classes
    - 1.1 Class_Name
    - 1.2 Class_State_Space
        - 1.2.1 Attr_Name
        - 1.2.2 Set
    - 1.3 Services
        - 1.3.1 Service_Name
        - 1.3.2 Input_Sets
        - 1.3.3 Output_Sets

1.3.4 Preconditions

1.3.5 Postconditions

2. R_I, which is the superclass relation

3. R_W, which is the whole/part relation

4. R_R, which is other relations

The structure in LISP which naturally matches the structure defined for classes is the class structure in the Common LISP Object System (CLOS). The classes were implemented as a CLOS class structure called *generic-class*. The attributes and services were also implemented as CLOS class structures because their structures also naturally matched the class structure. Each of the components of the OORA model correspond to a slot of the CLOS data structures.

The advantage to using CLOS class structures was that all components of each class are hidden inside the class. Therefore, there is no possibility of name clashes or confusion between the class names and the attribute and service names. The names of the attributes and services from one class can be used in another without any confusion as to where the attribute or service belongs. There cannot be two classes with the same name within any one model. There cannot be two attributes named the same within one class nor can there be two services with the same name within one class. Other than those restrictions, the names of classes, attributes and services can be repeated as other classes, attributes and services.

Detail on the method for accessing the components of each class will be discussed after the structures of the classes are discussed.

In the following sections, the CLOS class structure will be described, to be followed by the attribute class structure, the service class structure, and finally, the components of the attributes and services. These components are written using LISP's record structure, or *defstruct* structure.

CLOS Class Structure. Each class in the domain model is an instance of the following CLOS class, called *generic-class*. This class consists of a set of named slots. The number of slots in a given CLOS class and the contents of each slot are defined by the creator of the class based on the descriptors given to each slot. In the case of the class *generic-class*, there are eight slots defined. The *name* slot contains the name of the class. The "description" slot contains a description of the class. The *state-space* slots contains information on all attributes of a class. Each attribute is an instance of another CLOS class, described in the next section. The *services* slot contains information on all the services of a class. The services are also implemented as CLOS classes, and are defined later. The *inheritance* slot contains any parents of the class. The *whole-part* slot contains any whole-part relations for the class. The *relationships* slot contains any other relationships for the class. Finally, the *need-verified* slot contains "no" if the class has not been verified by the user and "yes" if it has.

Each slot can have zero or more slot options associated with it. These slot options provide mechanisms for customizing the slots, such as supplying default initial values, automatically generating functions for reading and writing slots, supplying initialization arguments used in instance creation, and supplying a documentation string for the slot.

Each slot in the *generic-class* class has been defined with particular slot options. The slot options used are the initial value for the slot when an instance of the class is created (:initarg), the name used to access the value of the slot (:accessor), and a documentation string that describes the function of the slot (:documentation). Examining *generic-class*, the initial value of the slot named *description* is set through the use of the function *:desc* when an instance class is first created. After the class is created, the value of the slot is accessed and changed through the use of the function of the name *desc*. The documentation string at the end of the class is not part of the class slot structure but is one

or more sentences used describe the *generic-class* structure. The following is the CLOS structure for the class *generic-class* which was used for each class in the model.

```
(clos:defclass generic-class ()
    ((name :initarg :name
           :accessor name
           :documentation "The name of the class:)
     (description :initarg :desc
                  :accessor desc
                  :documentation "A description of the class")
     (state-space :initarg :state-space
                  :accessor state-space
                  :documentation "The class state space)
     (services :initarg :services
               :accessor services
               :documentation "The class services")
     (inheritance :initarg :inheritance
                  :accessor inheritance
                  :documentation "The immediate superclasses")
     (whole-part :initarg :whole-part
                  :accessor whole-part
                  :documentation "The whole-part relation")
     (relationships :initarg :relation
                    :accessor relations
                    :documentation "Other relationships")
     (need-verified :initarg :verif
```

```
                :accessor verif
                :documentation "Does the class need user verification"
(:documentation "A generic class"))
```

In order to access any slot of an instance of *generic-class*, the instance must be referenced. For example, if One-Class is an instance of class *generic-class*, the name of One-Class would be accessed through the function (name One-Class). This function would return the name of One-Class. The value of the name of One-Class would be changed by using (setf (name One-Class) New-Name). This would set the value of the *name* slot of One-Class to New-Name. Using this class structure within CLOS therefore provides an encapsulation of the components of a class within the class. This avoids any conflicts caused by the names of attributes of different classes being the same, for example. The attributes of each class are only accessible through the class itself and are not stored as global names. This structure therefore embodies the object-oriented concept of encapsulation within a class of the class's attributes and services.

Using this CLOS structure also enables the capturing of all the components of the OORA mathematical model in one class structure. There is no requirement for separate inheritance, whole/part, other relationship, and message passing structures. All this information is contained in the CLOS class structure *generic-class*. This causes the information about whole/part and other relationships to be repeated in each of the classes involved in the relationship. The advantage is that there is only one structure to monitor compliance with the defined guidelines and rules and to ensure each class in the model remains consistent and complete throughout the model modification process. The inheritance, whole/part, other relationships, and message passing trees can be created through the use of the information in the CLOS class structure. This process was automated in the case of the message-passing tree and is discussed in a later section.

The following table relates the slots from the CLOS generic-class to the components of the OORA model.

Table 7-1
Relationship Between CLOS Class and OORA Model

OORA Components	generic-class slots
Class_Name	name
Class_State_Space	state-space
Services	services
R_I, which is the superclass relation	inheritance
R_W, which is the whole/part relation	whole-part
R_R, which is other relations	relationships

The *need-verified* slot is initially set to '(), or "no", stating that the user has not reviewed the requirements for the class as of yet. Once the user has verified the class, the value is set to true, or "yes".

CLOS Attribute Structure. Each attribute is an instance of the CLOS class called *attribute*. There are four slots defined. The *name* slot contains the name of the attribute. The *description* slot contains a description of the attribute. The *initial-value* slot contains any initial value required for the attribute when a new instance of this class is created. The default value for this slot is null, meaning no initial value is specified. The *a-set* slot contains the set of legal values for the attribute. The *need-verified* slot is false if the attribute has not been verified by the user and true if it has. The slots in the *attribute* class have the same options as the slots of the class *generic-class* except for the addition of a slot option, *:initform*, that is used to set the initial value for every instance of the class. For example, in the class *attribute*, the value of the *a-set* slot is initially set to an empty list

when an instance is created. This value can be set through the :initform option when the instance is created or the *:accessor* option after the instance is created.

The following is the CLOS attribute structure.

```
(clos:defclass attribute ()
    ((name :initarg :name
           :initform " "
           :accessor name
           :documentation "The name of the attribute.")
     (description :initarg :desc
                  :accessor desc
                  :documentation "A description of the attribute.")
     (initial-value :initarg :initial-value
                    :initform '()
                    :accessor initial-value
                    :documentation "Any initial value used when an object is created.")
     (a-set :initarg :a-set
            :initform '()
            :accessor a-set
            :documentation "The legal set of values.")
     (need-verified :accessor verif
                    :initform '())))
    (:documentation "A general structure for an attribute."))
```

The following table relates the components from the CLOS attribute structure to the components in the OORA model.

Table 7-2
Relationship Between CLOS Attribute Structure and OORA Model

OORA Components	**attribute slots**
Attr-Name	name
Set	a-set

CLOS Service Structure. Every service is an instance of the CLOS class *service*. There are four slots defined. The *name* slot contains the name of the service. The *description* slot contains a description of the service. The *input-set* slot contains information on the input parameters of the service. The *output-set* slot contains information on any output parameters of the service. The *preconditions* slot contains any preconditions for the service. The *postconditions* slot contains information on attributes that changed value as a result of the execution of the services and messages required for the service to do its function. The *need-verified* slot is false if the attribute has not been verified by the user, and true if it has. The slot options used in the service are the same as described for the class *attribute*. The following is the CLOS service structure.

```
(clos:defclass service ()
    ((name :initarg :name
           :accessor name
           :documentation "The name of the service")
     (description :initarg :desc
                  :accessor desc
                  :documentation "A description of the service")
```

```
(input-set :initarg :input-set
           :initform '()
           :accessor input-set
           :documentation "The output parameter list")
(output-set :initarg :output-set
            :accessor output-set
            :documentation "The output parameter list")
(preconditions :initarg :pre
               :accessor pre
               :documentation "The preconditions")
(postconditions :initarg :post
                :accessor post
                :documentation "The postconditions")
(need-verified  :initarg :verif
                :accessor verif
                :initform '()))
(:documentation "A generic service class"))
```

The following table relates the components from the CLOS service structure to the components in the OORA model.

Table 7-3
Comparison of CLOS Service Structure and OORA Model

OORA Components	**service slots**
Service_Name	name
Input_Sets	input-set
Output_Sets	output-set
Preconditions	preconditions
Postconditions	postconditions

Attribute and Service Components.

Attribute Values. The legal set of values for each attribute is described in the record structure in LISP defined by *defstruct*, shown later. This record structure is named *attrib* and consists of the components *base*, *lower* and *upper*. The *base* component is the base set; the *lower* and *upper* values are optional and have different meanings depending on the base set. The following table describes all combinations of base set and upper and lower values that are used by OAKS. The words in parenthesis match those used in the LISP implementation. These words for the legal base sets are not LISP defined types but only have meaning in the context of the OAKS system.

Table 7-4
Legal Sets for Attribute Values

Base Set (base)	**Lower Value (lower)**	**Upper Value (upper)**	**Comments**
Enumerated (enum)	A list of values		All the possible values are shown in the lower value.
Integer (int)	The lowest integer	The highest integer	The range of integers is optional.
Real (real)	The lowest real	The highest real	The real number are those with decimal points.
Character (char)	The lowest character	The highest character	The range of characters is optional.
String (str)	The lowest string (in alpha-numeric order)	The highest string (in alpha-numeric order)	The range of strings is optional.
Boolean (bool)			True or false.
Class (class)	a-class		The value is from the set of instances of a-class.
Attribute (attrib)	a-class	an-attribute	The value is from the legal set of values specified for an-attribute of a-class.
List of components			A list, each of whose components is the "attrib" record structure defined above.

The structures shown below are the LISP implementation of the table. The first structure, *proper-attr-setp* defines the permissible names that can be used for the base sets of attributes. These names are the same as those in the base set column of the table above. This establishes the basis for the second structure called *legal-set*, whereby a value is a legal-set type if it is one of the names contained in *proper-attr-setp*. The third LISP structure is the record structure for the legal set of values of an attribute. It shows three slots. The first contains a value that must be of type *legal-set* and has an initial value of int. The initial value is used only because LISP will not allow the type of a slot to be specified without an initial value. The second slot is the lower value and the third the upper value. There is no requirement, in general, to have values in these slots, so their default values are set to "none".

```
(defun proper-attr-setp (a-set)
 (or
  (eql a-set '())
  (eql a-set 'enum)
  (eql a-set 'int)
  (eql a-set 'real)
  (eql a-set 'char)
  (eql a-set 'str)
  (eql a-set 'bool)
  (eql a-set 'class)
  (eql a-set 'attrib)
  (listp a-set)))
```

```
(deftype legal-set ()
 '(satisfies proper-attr-setp))

(defstruct attrs
 (base 'int :type legal-set)
 (lower 'none)
 (upper 'none))
```

The aircraft maintenance and aircraft mission scheduling processes will be used to further illustrate the OAKS CLOS structure. The system contains the aircraft, aircrew, and maintenance (or support) personnel in a squadron. A squadron typically consists of a number of flights of aircraft. The aircraft contain parts, each of which is repaired by a certain repair shop. Maintenance personnel are assigned to a particular repair shop. Aircrew are qualified to fly certain aircraft. When an aircraft part needs maintenance, it may require the scheduling of space in a hangar for the repair as well as the scheduling of maintenance personnel qualified to repair the part. An aircraft mission requires the scheduling of aircraft, aircrew, and the range space in which the mission is flown.

To illustrate the use of the constructs, consider the class squadron. To create an attribute called *name* within the class the LISP construct would be:

```
(make-instance 'attribute
                    :name 'name
                    :desc "The name of the squadron."
                    :a-set (make-attrs :base 'str))
```

To create an attribute called *the-aircraft*, which is a list of all the aircraft in an object of class squadron, the LISP construct would be:

```
(make-instance 'attribute
                    :name 'the-aircraft
                    :desc "A list of the aircraft in the flight."
                    :a-set (make-attrs :base `(,(make-attr :base 'class
                                                           :lower 'aircraft))))
```

Input and Output Sets of Services. The members of the input and output sets are represented by a LISP *defstruct* called *parameterf* with slots of *name* and *values*.

```
(defstruct parameterf
 (name '())
 values)
```

Each member of the input set must contain the input parameter name in slot *name* and the legal set of values for that parameter in slot *values*. Each member of the output set contains just the legal set of values for the output parameter it represents. This is because output parameters are not named. The legal set of values for input and output sets can be one of the legal sets for attributes such as "int", the name of an attribute in the class which indicates the legal values are the same as for the named attribute, or the name of an attribute of another class. If the legal set of values is that of an attribute of another class, the set of values is a list of the form (:a class-name attr-name). If the legal set of values is an instance of another class, the set of values is a list of the form (:c class-name).

The ":a" and ":c" notation used to identify the legal set of values for the input and output sets was necessary to distinguish between the names of attributes within the class

and the names of attributes of other classes. It is possible for the name of an attribute to be the same as the name of another class in the model or an attribute in another class.

For example, an input set consisting of one parameter whose name is *symptoms* and whose legal set of values is that for the attribute *legal-symptoms-list* of class *repair-symptoms*, would use the following structure:

```
:input-set `(,(make-parameterf :name 'symptoms
                               :values '(:a repair-symptoms legal-symptoms-list)))
```

The services are modeled as functions and return a single value. This value can be a single element or a list of elements. Each output parameter is therefore defined solely by its set of legal values and no name is required.

For example, an output set consisting of one parameter whose value is the name of an object of class *aircraft* would use the following structure:

```
:output-set `(,(make-parameterf :values '(:c aircraft)))
```

Preconditions of Services. The preconditions were implemented in LISP as an expression that evaluates to true or false. An example of a precondition is as follows:

```
'(not (member new-flight flights))
```

Postconditions of Services. The postconditions are represented by a LISP *defstruct* record structure with possible *defstruct* structures embedded.

The basic postcondition is as follows:

```
(defstruct postf
 (atts '() :type list)
 (messages '() :type list))
```

The *atts* slot of *postf* consists of structures that represent a list of attributes of the class that have possibly changed as a result of the service call, and the value of those changed attributes upon leaving the service. This *atts* slot consists of a list of *defstruct* structures that contain the attribute name and the new attribute value.

The following is the structure for the list of attributes that have changed as a result of the service call:

```
(defstruct attr-val
 name
 value)
```

The *name* is the name of an attribute of the class. The *value* is a free-form structure that shows the new value of the attribute. The new value of the attribute can be "changed", indicating that an exact value cannot be determined but the attribute may be changed as a result of the service call.

The *messages* slot of *postf* represents the services of other classes used by this service. This information is represented as a list of pairs. Each pair is a list consisting of the class and the service.

Again, as an example, a postcondition that adds a new-person to an aircrew list and sends a message to create a new instance of aircrew would be represented by:

```
:post
    (make-postf :atts `(,(make-attr-val :name aircrew
                                        :value '(cons new-person aircrew)))
                :messages '((aircrew create))) )
```

Inheritance. The inheritance structure of the model is contained in the information in the *inheritance* slot of each class. The *inheritance* slot of each class contains a list of parents of the class. If the class has no parents, the inheritance slot is empty. For example, if the *inheritance* slot of the class *aircrew* contains the list "(personnel)", this indicates that *personnel* is the only parent of *aircrew*.

Relationships. Both whole/part and other relationships are represented by the *defstruct* structure as follows:

```
(defstruct relation
 (name 'whole/part)
 class1
 range1
 class2
 range2)
```

The default for the name is "whole/part". If *relation* is used to represent a whole/part structure, class1 is the whole and class2 is the part. The whole/part relation is shown in both the whole and the part classes as structures in the whole-part slot of both classes. This provides a means to easily trace effects of changes on any class. The whole/part relations are checked when they are changed, removed or added by OAKS to ensure the system remains consistent. For example, when a whole/part relation is deleted in one

class, it is automatically deleted in the other class involved in the whole/part relation. Other checks made on whole/part relations are discussed in later sections on the implementation of the guidelines and rules and the model modification process.

An example of a whole/part relationship is as follows:

```
:whole-part `(,(make-relation :class1 'squadron
                              :range1 '(1 n)
                              :class2 'flight
                              :range2 '(1 1))
              ,(make-relation :class1 'flight
                              :range1 '(1 n)
                              :class2 'aircraft
                              :range2 '(1 1)) )
```

This is part of the class *flight*. This shows **flight** has between 1 and n *aircraft* as a part and that it is part of only one *squadron*. These whole/part relations will also be contained in the *squadron* and the *aircraft* classes.

Other relationships are handled in a similar manner. The relationship is named with the two part name as previously described, and that same name is used in both classes involved in the relationship.

An example of other relationships is as follows:

```
:relation `(,(make-relation :name 'has-a/for-a
                            :class1 'aircraft
                            :range1 '(1 1)
                            :class2 'aircraft-schedule
                            :range2 '(1 1)))
```

Aircraft has-a aircraft-schedule and aircraft-schedule is for-a aircraft. This relation will be shown exactly as above in both the "aircraft" and the "aircraft-schedule" class. Therefore, the relation must be inserted into both classes.

Example of a Class Structure. The OAKS model consists of a set of classes. Each class is an instance of the CLOS class "generic-class". The attributes are instances of the "attribute" class and the services of the "service" class. The following is an example of the "plans-and-scheduling" class:

```
(setf plans-and-scheduling
  (let*
    ((range
     (make-instance 'attribute
            :name 'range
            :desc "The range schedule."
            :a-set (make-attrs :base 'class
                        :lower 'range-schedule) ))

    (missions
     (make-instance 'attribute
            :name 'missions
```

```
        :desc "The missions that have been scheduled."
        :a-set (make-attrs :base `(,(make-attrs :base 'class
                                    :lower 'mission))) ))

(mission-request
 (make-instance 'service
        :name 'mission-request
        :desc "A request for the scheduling of a mission."
        :input-set `(,(make-parameterf :name 'ac-list
                        :values '((:c aircraft)
                                  (:a aircraft configuration)))
                ,(make-parameterf :name 'list-of-aircrew
                        :values '((:c aircrew)))
                ,(make-parameterf :name 'duration
                        :values '(:a schedule-event duration))
                ,(make-parameterf :name 'range-info
                        :values '(:a mission range-info)))
        :output-set '()
        :pre '()
        :post
        (make-postf :atts `(,(make-attr-val
                        :name 'missions
                        :value '(cons new-mission missions)))
                :messages '((aircrew get-sched)
                        (aircraft get-sched)
                        (mission create)
```

```
                    (aircraft-schedule add-mission)
                    (aircrew-schedule add-mission)
                    (range-schedule add-mission))) ))

(mission-complete
 (make-instance 'service
        :name 'mission-complete
        :desc "A mission has been completed."
        :input-set `(,(make-parameterf :name 'the-mission
                            :values '(:c mission))
                ,(make-parameterf :name 'hours
                            :values '(:a mission ac-info))
                ,(make-parameterf :name 'crew
                            :values '(:a mission aircrew-list))
                ,(make-parameterf :name 'date
                            :values 'int)
                ,(make-parameterf :name 'time
                            :values '(:a mission time)))
        :output-set '()
        :pre '(member mission missions)
        :post
        (make-postf :messages '((aircraft configuration)
                            (aircraft-part update-flight-hours)
                            (aircrew update-hours)
                            (mission change-date)
                            (mission change-time)
```

```
                    (mission change-ac-info)
                    (mission change-status))) ))

(cancel-mission
 (make-instance 'service
        :name 'cancel-mission
        :desc "A mission is canceled."
        :input-set `(,(make-parameterf :name 'the-mission
                        :values '(:c mission)))
        :output-set '()
        :pre '(member the-mission missions)
        :post
        (make-postf :messages '((aircraft-schedule remove-mission)
                    (mission get-date)
                    (mission get-duration)
                    (mission get-config)
                    (aircrew-schedule remove-mission)
                    (mission get-mission-type)
                    (range-schedule remove-mission)
                    (mission get-aircraft)
                    (mission get-range-info)
                    (mission change-status))) )) )

(make-instance 'generic-class
        :name 'plans-and-scheduling
        :desc "Schedule missions."
```

```
:state-space (list range missions)
:services (list mission-request mission-complete
          cancel-mission)
:inheritance '()
:whole-part '()
:relation `(,(make-relation :name 'uses/used-by
                    :class1 'plans-and-scheduling
                    :range1 '(0 n)
                    :class2 'mission
                    :range2 '(1 1)))) ))
```

The entire class contains two attributes, named *range* and *missions*, and three services, named *mission-request*, *mission-complete*, and *cancel-mission*. These show local attributes and services only and not those possibly inherited from other classes. In this example, the inheritance slot is an empty list, signifying this class does not inherit from any other class.

The set of values of the attribute named *range* is all objects of the class *range-schedule*. The set of values of the attribute named *missions* is all lists of objects of class *mission*.

The service *mission-request* has four input parameters that are shown in the input-set. This service does not return a value; this is indicated by the empty list in the output set. The first parameter, *ac-list*, is a list of pairs consisting of objects of class *aircraft* and values of type of attribute *configuration* in the class *aircraft*. The second parameter, *list-of-aircrew*, is a list of objects of class *aircrew*. The third parameter is a single value of the type of the attribute *duration* of class *schedule-event*. And the fourth parameter, *range-info*, is a single value of the type of the attribute *range-info* of class *mission*.

The service *mission-request* possibly modifies one local attribute, *missions*, by changing its value to "(cons new-mission missions)". This is shown in the *atts* slot of the postcondition in the *:post* slot. The service *mission-request* also uses four services of classes outside this class. This is shown in the *:messages* slot of *:post*. The service of *aircraft* named *get-sched* is used, along with the *create* service of *mission*, the *add-mission* service of *aircraft-schedule*, the *add-mission* service of *aircrew-schedule*, and the *add-mission* service of *range-schedule*.

The plans-and-scheduling class has a relation to the class *mission* as shown in the *:relation* slot of the class.

Entire Model. The entire model is represented by the global list, *list-of-classes*. This list contains the classes as represented by their CLOS generic-class structure. Set_Of_Classes in the domain model is represented by *list-of-classes* in OAKS.

This section established the LISP structures used to implement the defined OORA mathematical model components and relationships. The OAKS domain model consists of the global list *list-of-classes*, each element of which in an instance of the class "generic-class". The OAKS domain model represented in *list-of-classes* is the basis for the analysis done by the guidelines and rules and the modifications performed to create the problem model. It is this *list-of-classes* that is analyzed so the modifications made to it keep the model consistent and complete with respect to the defined guidelines and rules.

The next section first defines additional guidelines and rules not defined in the previous chapter. These additional guidelines are rules are based on the LISP structures that make up the OAKS domain model. The section then discusses how the domain-independent guidelines and rules defined in the previous chapter are implemented in OAKS.

Domain-Independent Guidelines and Rules

Structure-Based Guidelines and Rules. Chapter 6 defined domain-independent guidelines and rules that were based on existing OORA methods. These guidelines and rules are independent of the code structure of the domain model within OAKS. This section discusses and analyzes guidelines and rules that are based on the required characteristics of the LISP code structure in OAKS. These guidelines and rules could not be developed until the code structure for the classes was developed as was done in the previous section. An example of a structure-based rule is that any classes used in the a-set slot of an attribute must exist in the model.

The following sub-sections define each structure-based guideline and rule, first using an English description and then using first-order predicate logic.

Classes. All class names must be unique within a single model. This requirement is implemented in the LISP function unique-class-names.

$$\forall a,b \in \text{*list-of-classes*}\ [(a \neq b) \Leftrightarrow (\text{name } a) \neq (\text{name } b)]$$

Attributes. For all classes in the "a-set" slot of attributes, there must exist a class in *list-of-classes*. This requirement is implemented in the LISP function model-att-class-check.

$$\forall c \in \text{*list-of-classes*}, \forall a \in (\text{state-space } c), \forall x \in (\text{a-set } a)$$
$$[((x.\text{base} = \text{class}) \vee (x.\text{base} = \text{attrib})) \Rightarrow (x.\text{lower} \in \text{*list-of-classes*})]$$

For all attributes and their classes in the "a-set" slot of attributes, there must exist an attribute of that name in that class. This requirement is implemented in LISP function model-att-att-check.

$$\forall c \in \text{*list-of-classes*}, \forall a \in (\text{state-space } c), \forall x \in (\text{a-set } a)$$

$$\{ (x.\text{base} = \text{attrib}) \Rightarrow [(x.\text{upper} \in \text{*list-of-classes*}) \ \Lambda \ (x.\text{lower} \in (\text{attrs } x.\text{upper}))]\}$$

Services. For all services in all classes, for all input sets, if there are values that are instances of a class, that class must be in the *list-of-classes*. This requirement is implemented in model-serv-att-check.

$$\forall c \in \text{*list-of-classes*}, \forall s \in (\text{services } c), \forall x \in (\text{input-set } s)$$

$$\{[((\text{first } (x.\text{values})) = :c \ \text{V} \ ((\text{second } (x.\text{values})) = :a)] \Rightarrow$$

$$[\text{second}(x.\text{values})) \in \text{*list-of-classes*}]\}$$

For all services of all classes, for all input sets, if there are values whose types are attributes of other classes, those attributes must exist in that class. This requirements is implemented in model-serv-att-check.

$$\forall c \in \text{*list-of-classes*}, \forall s \in (\text{services } c), \forall x \in (\text{input-set } s)$$

$$\{[\text{first } (x.\text{values}) = :a] \Rightarrow [\text{third } (x.\text{values}) \in \ (\text{state-space } (\text{second } (x.\text{values})))]\}$$

For all services of all classes, for all attribute/value pairs in the postconditions, the attributes must exist in the class. This requirements is implemented in model-serv-att-check.

$$\forall c \in \text{*list-of-classes*}, \forall s \in (\text{services } c), \forall p \in (\text{post } s), \forall a \in (p.\text{atts})$$

$$[p.\text{value} \in (\text{state-space } c)]$$

For all services of all classes, for all messages in the postconditions, the classes and their services must exist. This requirement is implemented in model-serv-att-check.

$$\forall c \in \text{*list-of-classes*}, \forall s \in (\text{services } c), \forall p \in (\text{post } s), \forall m \in (p.\text{messages})$$
$$\{[(\text{first } m) \in \text{*list-of-classes*}] \wedge [(\text{second } m) \in (\text{services } c)]\}$$

Whole/Part. For all classes, if there exists a whole/part structure in one class, it must exist in the other. This requirement is implemented in model-wp-check.

$$\forall c \in \text{*list-of-classes*}$$
$$\{[w \in (\text{whole-part } c) \wedge ((d = w.\text{class1}) \vee (d = w.\text{class2})) \wedge (c \neq d)] \Rightarrow w \in (\text{whole-part } d)\}$$

Relationships. For all classes, if there exists a relationship in one class, it must exist in the other class involved in the relationship. This requirement is implemented in model-rel-check.

$$\forall c \in \text{*list-of-classes*}$$
$$\{[r \in (\text{relation } c) \wedge ((d = r.\text{class1}) \vee (d = r.\text{class2})) \wedge (c \neq d)] \Rightarrow r \in (\text{whole-part } d)\}$$

Inheritance. For all classes, any parents in the inheritance slot must exist in the model. This requirement is implemented in model-parent-check.

$$\forall c \in \text{*list-of-classes*}$$
$$[i \in (\text{inheritance } c) \Rightarrow i \in \text{*list-of-classes*}]$$

General OORA Guidelines and Rules. These are the domain-independent guidelines and rules defined in chapter 6 that are used by OAKS to evaluate the domain and problem models. Each guideline and rule used is identified by the same numbering scheme used in chapter 6.

Class.

GR1

Class names should be singular nouns. To evaluate the use of singular nouns without the use of a parser, OAKS finds classes that have "s" endings on their name. These classes are not necessarily named wrong, but they should be flagged and shown to the user. For example, the class name "dress" ends in "s" but is a singular noun. The LISP function singular-noun-check takes one class and returns true if it is not a singular noun. The procedure model-singular-noun-check evaluates the entire model for class names that end in "s" and returns a list of those names.

GR3,8,9,10,13

OAKS determines which classes have no connection to other classes. A class can be connected to other classes in one of five ways:

- It is the parent of another class.
- It is part of a whole-part relationship.
- It is part of a general relationship.
- It is called by another class through a message connection.
- It calls another class through one of its services.

The function connectionp returns true if the class is connected to other classes. The function unconnected-classes evaluates the entire model and returns a list of classes that are unconnected.

GR12

The name of any new class is evaluated by OAKS to determine if it is related to an existing class name. This is crudely done by looking for the first dash (if any) in the class name and matching the remainder of the name with the full names of existing classes. For example, "an-aircraft" would match with "aircraft". "any-old-aircraft" would match with "old-aircraft" but not with "aircraft". This identifies a possible inheritance link between classes.

The function class-name-match takes a new class name and returns any existing class names that match it.

GR15

The description of a class contains information on what the class represents, its general properties, and what processing is required other than that for updating attribute values. The description is contained in the description slot of each class.

Inheritance.

GR3

OAKS looks for attributes that are the same in different classes. These classes may share an inheritance structure. This is checked when the domain model is entered. An attribute contains slots for its name, description, and the valid set for its values. The name and description are not good for comparison. A different name could be used to represent the same attribute, and the description would not be exactly the same. Therefore, the comparison is made on the legal set of values. OAKS checks to determine if 80% or more of the attributes of the new class share the same legal values for each attribute as attributes in another class. The 80% was arbitrary and can easily be changed in the code. If another class matches 80% of the new class's attributes, the class is brought to the user's attention. It is possible an existing class is related to the new class through inheritance.

This check does not guarantee there is any relationship between the two classes: it merely identifies a possible relationship.

The function similar-atts returns a list of other classes that have similar attributes. The function model-similar-atts examines the entire model and returns a list of class names that have similar attributes.

GR6

Do not nest subclasses deeper than three levels. If the level of the bottom-most class in an inheritance structure is considered level 0, then if any of the superclasses are at level 3 or above, the model should be examined for changes. The check of the depth level of all classes in the model is recommended as one of the checks the creator of the domain model should conduct. It is also one of the tests run when the user asks for issues that are advisory (i.e., they do not have to be resolved before the model is considered complete). These advisory issues are discussed in depth in a later section.

The function class-depth returns the level of a class. If the child class is at level n the parent is defined to be at level n+1. It allows for multiple inheritance by returning the maximum class depth. The function model-class-depth returns a list of pairs of class names and the class level.

GR9

There should be at least two subclasses per superclass.

The function two-subclass-check returns the name of the current class if it has only one child. The function model-two-subclass-check checks the entire model and returns superclasses that have only one child.

State Space.

GR5

See Inheritance GR3.

Services.

GR2,3,4,5,6,8,9,11,12,13,15,16,17

Four service templates were developed as LISP functions. The templates either return the value of an attribute or change the value of an attribute. The use of a template greatly simplifies the creation of a service by automatically filling in many of the service slots based on knowledge of the functioning of the service. The following are descriptions of the functions that implement the four service templates.

1. Change-att-template

This function is used to create a new service. This new service changes the value of an attribute if the attribute consists of a single value. If the value of the attribute is a list of values, either remove-element-template or add-element-template is used. The input parameters for this function are the new service's name and the name of the attribute whose value is changed by the new service. When the function creates the service, it automatically fills in the values for the new service's description, input set, output set, precondition and postcondition.

2. Return-att-template

This function is used to create a new service that returns the value of an attribute. The input parameters for this function are the new service's name and the name of the attribute whose value is returned by the new service. When the function creates the service, it automatically fills in the values for the new service's description, input set, output set, precondition and postcondition.

3. Add-element-template

This function is used to create a new service. This new service adds an element to an attribute for attributes that consist of a list of values. The input parameters for this function are the new service's name and the name of the attribute whose value is changed by the new service. When the function creates the service, it automatically fills in the

values for the new service's description, input set, output set, precondition and postcondition.

4. Remove-element-template

This function is used to create a new service that removes an element from an attribute for attributes that consist of a list of values. The input parameters for this function are the new service's name and the name of the attribute whose value is changed by the new service. When the function creates the service, it automatically fills in the values for the new service's description, input set, output set, precondition and postcondition.

GR7

The messages are traced through the model using the procedure trace-messages. An initial class and service name are given, and a trace of the message connections is output.

GR10

The function message-connectionsp takes a class and returns true if it either has message connections with another class or is a parent of another class. The function model-message-connectionsp examines all classes in the model and returns a list of those that do not have message connections or are not parents.

GR14

The function similar-servs takes a class name and returns a list of classes whose services match at least 80% of the services of the original class. A service matches another service if the values slots of the input set and output set are equal. The name of the service is arbitrary and would not be a good basis for comparison. The same argument holds for the names of the input parameters. The values slots of the parameters represent the legal set of values those parameters can take on. If the values slot contains the name of a local attribute, the legal set of values for the local parameter is used as a basis for comparison, and not the attribute name, which is arbitrary.

The services of the input class (class A) are compared to the set of services of each class in the model. If the services are being compared to the services of class B and if a service of class A matches a service of class B, the matching service of class B is marked and is not used for comparison again. This ensures one service of class B does not match every service of Class A. If 80% or more of the services of Class A matches those of Class B, Class B is considered possibly related to class A.

The function model-similar-servs looks at all classes in the model and returns a list of class names and the classes possibly related to it.

Whole Model.

GR1

The function one-attributep takes a class name and returns true if the number of attributes is less than two. The function model-one-attributep returns a list of all classes in the model which have less than two attributes.

GR2,10

The function one-servicep takes a class name and returns true if the number of services is less than two. The function model-one-servicep returns a list of all classes in the model which have less than two services.

Instead of using a *create* service to show default values for the attributes of a class, a *default-value* slot was used in the attribute structure. By default, this value is set to '(), which means the attribute value is empty when a new object of that class is created. If any other value is needed, the attribute would override the default with a value other than '() in the *default-value* slot.

GR6

The function num-att-ser returns the total number of attributes and services in a class. The function model-num-att-serv returns a list of class names and the number of attributes and services in each class. Model-ave-att-serv returns the average number of attributes

and services in the model. This is used in the evaluation of the domain model. If a new class has 20% or more than the average, this will be brought to the attention of the analyst to determine if it can be broken into smaller classes.

GR7

The function share-att-serv takes a class and returns any classes that share 80% of its attributes and 80% of its services. The function model-share-att-serv looks at the entire model for any classes that share 80% of their attributes and services with another class.

Model Evaluation. Before the user can use and modify the domain model to fit a particular problem, the domain model must adhere to the guidelines and rules outlined above. Some of the guidelines and rules are required in that they must be adhered to before the model is used. Some of the guidelines and rules are advisory, in that they can be violated and the problem model would still be valid.

These model evaluation functions proved extremely useful during the development of the domain model. Using these functions, many errors in the domain model were uncovered and easily identified for correction. Originally, these evaluation functions were envisioned solely for use during creation of the problem model. It became apparent that these were as useful, if not more useful, to the developer of the initial domain model. These functions can be used on any domain model in OAKS since they are totally domain-independent.

The following are LISP functions that must run successfully before the model can be used. A successful completion returns a null result.

1. unique-class-names ()

   Ensures the class names within a model are unique.

2. model-att-class-check ()

   Evaluates all the attributes to ensure any classes used in the a-set slot of an attribute exist in the model.

3. model-att-att-check ()

Evaluates all the attributes to ensure any attributes, external or internal, used in the a-set slot exist.

4. model-serv-att-check ()

Evaluates all the services to ensure any classes, attributes or services used are valid.

5. model-wp-check ()

Examines all whole/part structures to ensure they are repeated in their respective class.

6. model-rel-check ()

Examines all other relationships to ensure they are repeated in their respective class.

7. model-parent-check ()

Checks all parents to ensure they exist in the model.

8. model-input-set-names ()

Checks all input set names of services for validity.

9. model-relation-classes-different ()

Ensures that the two classes in relation and whole-part structures are different.

10. model-remove-repeated-relations ()

Removes any repeated relations in a class.

11. model-remove-repeated-messages ()

Removes any repeated messages in a class.

12. model-unique-att-names ()

Evaluates attribute names to ensure they are unique within a class and are not equal to the names of valid attribute values.

13. model-unique-serv-names ()

Evaluates services names to ensure they are unique within a class.

These 13 mandatory requirements are evaluated as the model is changed. In some cases the model is not allowed to be changed if one of these 13 are violated. For example, an attribute that has the same name as an existing attribute cannot be added to a class. In other cases, the change is allowed but an entry is added to a global list called *pending-issues* that represents a problem that must be fixed before the model is complete. The *pending-issues* list must be null before the model is considered complete. An example of a change that would cause entries in *pending-issues* would be the addition of a relation where one class in the relation does not yet exist in the model. What type of entries are allowed in *pending-issues* and when they can be removed are discussed later.

Some requirements are advisory in that they point to a possible, but not definite, problem in the model. For example, a class name normally should not end in "es" because class names should be singular nouns. But a class name of "bus" would be legal. Until a better parser is added to OAKS, this crude check of "s" at the end of a name can be violated.

The following are the advisory guidelines and rules

1. unconnected-classes ()

   Returns all classes that are unconnected to any other class in the model.

2. class-name-match ()

   Determines if a class has a possible relation to another class in the model by examining its name.

3. model-singular-noun-check ()

   Returns all class names that end in "s".

4. model-class-depth ()

   Returns a list of class names and their depth in the inheritance tree.

5. model-two-subclass-check ()

   Returns all classes that are parents that only have one child.

6. model-similar-atts ()

   Returns all classes that have similar attributes.

7. model-similar-servs ()

   Returns all classes that have similar services.

8. model-one-attributep ()

   Returns all classes that have one or zero attributes.

9. model-one-servicep ()

   Returns all classes that have zero or one service.

10. model-share-att-serv ()

    Returns classes that share 80 percent of their attributes and services.

These ten guidelines and rules are not evaluated each time the model is changed, but at the user's request or before the model is considered complete. Any violations of these rules are brought to the user's attention, but the user is not required to adhere to any of them. Any violations of these advisory guidelines and rules are captured in a list called *advisory-issues*, which is discussed in more detail later.

This section defined new structure-based guidelines and rules and defined and analyzed how these new guidelines and rules and the domain-independent guidelines and rules defined in Chapter 6 were implemented in OAKS. These guidelines and rules are used to define consistency and completeness in OAKS and therefore are used to flag the user when the problem model becomes inconsistent or incomplete. The LISP functions that implement these guidelines and rules are used to evaluate the initial domain model and the evolving problem model. The results of the evaluation are shown to the analyst who develops the initial domain model and to the user who develops the problem model. These results are either shown as issues that must be resolved prior to the model being considered complete or as advisory issues that may or may not need to be addressed.

Domain-Dependent Guidelines and Rules

Chapter 6 defined a set of domain-dependent guidelines and rules. This section implements and analyzes this last set of guidelines and rules.

The domain-dependent information in OAKS is represented by the associative list called *necessary-classes*. This list contains those classes and any of its attributes and services that are necessary to the completed model and therefore cannot be deleted from the model. However, the names and components of these classes, attributes and services can change, allowing the user to adapt these structures to the problem being solved. The classes in the model are the most likely not to change from one problem in the domain to another; hence they are the most likely to be required in a domain. For example, if the domain is an aircraft maintenance squadron, the "aircraft" class would be required in all problems in the domain. There would also be an attribute that would represent some identification of the aircraft, such as the tail number. The class and the attribute would be required, even though the name, "tail-number" could change if that terminology is not used in the problem.

Relationships are more likely to change than classes and therefore were not included in the *necessary-classes*. Also, inheritance was not included, because classes that are parents are not allowed to be deleted from the model. These restrictions could be added if deemed necessary for a particular domain. For purposes of illustration of the concept of domain-dependent guidelines and rules, the most likely constant structures were chosen for implementation.

The list *necessary-classes* is in the form of a list of sublists. Each sublist contains the class name, a possibly null list of attributes that cannot be deleted, and a possibly null list of services that cannot be deleted. An example of the LISP construct is as follows:

```
(defparameter *necessary-classes*
    '((aircraft (tail-number) ())
      (aircrew () (get-sched))
      (aircraft-part () ()))
```

This shows that the classes of aircraft, aircrew and aircraft-part cannot be deleted. In addition, the attribute tail-number of aircraft and the service get-sched of aircrew cannot be deleted.

This section discussed the implementation of the last set of guidelines and rules. The OAKS system now contains guidelines and rules based on general OORA principles and methods, based on the OAKS domain model structure, and based on the domain itself. These implemented guidelines and rules form the code for the model evaluation portion of OAKS.

Problem Model Modifications

Overview. This section defines the allowed changes to the evolving problem model. The LISP functions that implement these allowed changes use the code in the model evaluation portion to determine if a change will cause the model to become inconsistent or incomplete. If the model does become inconsistent or incomplete, the model modification code must determine whether to allow the change and handle the problem using a pending issues or an advisory issues, or to not allow the change. These issues and how they were handled in the OAKS implementation are discussed in this section.

The problem model is created by changes to the domain model. Not all changes are allowed, and a few changes must occur in a certain order. For example, a class that is a parent cannot be deleted, and an attribute cannot be added unless the class it is part of is in

the model. However, most of the changes are not required to be accomplished in any particular order. The changes to one class do not have to be complete before changing another class, for example. Another example is adding a whole/part relation to an existing class. A whole/part relation is a relation between two classes. The other class of the relation does not have to exist in the model in order to add the whole/part relation to an existing class. All that is required is that one class in the relation exist. This allows for great flexibility in creating the problem model and allows the user to revisit any portion of the model as many times as desired.

The problem model is complete when all issues in the *pending-issues* list have been satisfied, i.e., when the *pending-issues* list is empty. When a user first starts to create a problem model, the *pending-issues* list starts with an entry indicating that all the classes in the model have not been verified. This forces the user to at least examine the structure of each class, attribute and service. This issue is discussed in detail later in this section. As changes are made to the model, issues may be added or removed from the *pending-issues* list.

The LISP functions that implement the model modification process use the evaluation functions discussed in the previous section to determine if a change has violated any guidelines and rules.

The following sections discuss the changes that can be made to each component of the model, the possible modifications to the *pending-issues* list based on the changes, and the LISP functions used to implement the changes. When a LISP function calls another LISP function, the functions that are called are shown indented under it.

Change the Name of a Class. Because all the class names must be unique, the new name must first be checked to see if it is already the name of another class. If it is not, then the class name can change. This involves changing the class itself, the name of the

class in the list-of-classes, wherever that class name is used in attributes and services, and the name in inheritance and relation slots of the class itself and other classes.

If the old class name is used in any entries in *pending-issues*, it is changed to the new name. Also, a new class name may resolve some entries in pending issues. For example, if there is a class name used in the a-set slot of an attribute that was nonexistent, there would be an entry in *pending-issues*. If the new class name is the same as the nonexistent class, the entry would be removed. Also, when there is a relation created, one of the classes in the relation may not exist at the time the relation is created. This would add an entry in *pending-issues* on the class that is missing and the relation that must go into the class. If the new class name matches this missing class name, the relation is added and the entry in *pending-issues* is deleted.

The name of the class is also changed in *necessary-classes*, if the class is in that list.

Entries that can be removed from *pending-issues* are:

(1) (atts-classc class-name att-name)

The a-set slot of att-name contains the new class name

(2) (atts-attc class-name att-name)

The a-set slot of att-name contains the new class name

(3) (check-parameter class-name service-name input-set parameter)

The parameter contains the new class name

(4) (check-parameter class-name service-name output-set parameter)

The parameter contains the new class name

(5) (check-messages class-name service-name message)

The message contains the new class name

(6) (missing-class-and-relation class-name the-relation)

The class-name is the same as the new class name.

LISP functions used are:

change-class-name

change-class-name-slot

change-name-in-class

change-class-in-inheritance

change-class-in-relations

change-class-name-in-pending

new-class-pending

add-rel-to-new-class

change-name-in-class

change-name-in-atts

change-name-in-servs

change-class-name-in-pending

change-class-in-io-parameter

change-class-in-rel

Change the Description of a Class. Changing the description simply requires replacing the old description with the new. There are no checks to be made, except that the new description must be a string.

LISP functions used are:

change-class-desc

Change the Name of an Attribute. The attribute names must be unique within a class, but they can be the same as an attribute name in another class. First, the attribute name is checked to see if it is unique in the class and that it is not the same as the name of one of the legal attribute sets. The legal attribute sets are enum, int, real, char, str, bool, class, and attrib. The name cannot be the same as a legal attribute set because it would cause confusion if it is used in the input set or output set of a service. The values of the input or output parameters can be an attribute name of a local attribute, an attribute of

another class, or one of the legal set of attribute values. If it is a local attribute name, OAKS checks to ensure the name is one of the attributes of the class. Therefore, the attribute names cannot be one of the names of the legal set of attribute values.

If the attribute name passes these checks, the name is changed in the local slot and the local services. The name must also change globally, in the attributes and services of other classes. An attribute of another class may use this attribute as its value. If there are classes or attributes of other classes with the same name as the changed attribute name, these names are not modified. OAKS marks all attribute names so it is known what class the attribute belongs to.

If the old attribute name is used in *pending-issues*, the old name must be changed to the new name. Any entries in *pending-issues* resolved by changing the attribute name are removed. For example, there may be an input parameter that references a nonexistent attribute name that may be the new name of the attribute.

If the old name is used in *necessary-classes*, it is changed there.

Entries that can be removed from *pending-issues* are:

(1) (atts-attc class-name attribute-name)

The attribute-name is the same as the new attribute name and in the same class.

(2) (check-parameter class-name service-name input-set parameter)

The new attribute name is used in the parameter.

(3) (check-parameter class-name service-name output-set parameter)

The new attribute name is used in the parameter.

(4) (check-attr-val class-name service-name att-name)

The new attribute name is the same as att-name.

LISP functions used are:

change-att-name

proper-attr-setp

change-att-name-in-atts

change-att-name-in-servs

change-att-name-in-pending

remove-missing-att-entries

change-att-name-in-atts

att-name-sub

change-att-name-in-servs

change-att-io-set

change-att-post

Change the Description of an Attribute. Changing the description simply requires replacing the old description with the new. There are no checks to be made, except that the new description must be a string.

LISP functions used are:

change-att-desc

Change the A-Set Slot of an Attribute. The new value consists of a base and optional lower and upper values. If the base is a list, there are no lower or upper values. In this case, the a-set value is a list of elements. Each element of the list is a sublist made of a base and optional upper and lower values. The sublist represents the structure of each element of the list.

The new value structure is checked to ensure it is proper. If the base value is not a list, it must satisfy proper-attr-setp, i.e., it must be one of the legal set of attribute values. If the base value is a list, the base values of each of the sublists must satisfy proper-attr-setp. For any base value (whether the value structure is an atom or the value structure is a list), if the value is "class" there must be a lower value. If the value is "attrib", there must be a lower and upper value.

If the structure of the value is correct, the attribute a-set is changed in the local slot of the attribute. The a-set value is not used outside the attribute, except through the attribute name, so no replacements are made outside the attribute.

System-wide checks must be made based on the changes. First, the LISP function atts-classc examines the attribute (not the name - the attribute CLOS class structure) and ensures all classes used as a basis for attribute sets, either as classes or classes and attributes, are members of the set of classes. A null result means all classes are members of the set of classes. If the result is not null, that means there is a class used that is not a member of the set of classes. In this case, a list consisting of (atts-classc class-name att-name) is added to the list of *pending-issues*. This means the missing class must be added before the model is complete. The class name may not have been added yet by the user and may be added later. If atts-classc result is not null, atts-attc automatically fails because if the class does not exist, the system cannot check to see if the attribute exists within the class. If the result of atts-classc is null, atts-attc is run to check to see if any attributes of other classes used as a basis for the attribute are attributes of that class. If the result of atts-attc is not null, the list (atts-attc class-name att-name) is added to the list of pending issues.

If either of the system-wide checks passes, the *pending-issues* list is checked to see if any entry matches either (atts-classc class-name att-name) or (atts-attc class-name att-name). If there is a match, the matching entry is removed because now the test has passed successfully. It may be that the user has changed the structure to remove the problems.

A list is not added to pending issues if it is already on the list. For example, a class may be used more than once as a basis for an attribute, but the test only needs to pass once to prove the class now exists.

Entries that can be removed from *pending-issues* are:

(1) (atts-classc class-name attribute-name)

(2) (atts-attc class-name attribute-name)

If the old a-set slot contained classes and/or attributes that do not exist in the model, there are entries in *pending-issues*. If the new a-set slot is valid, the entries are removed.

Entries that can be added to *pending-issues* are:

(1) (atts-classc class-name attribute-name)

The class that is used in the new a-set slot does not exist.

(2) (atts-attc class-name attribute-name)

The class and/or attribute that is used in the new a-set slot does not exist.

LISP functions used are:

change-attr-a-set

create-attrs-structure

Change the Initial Value of an Attribute. An attribute can have an initial value that the attribute takes on when a class containing that attribute is first created. The initial value is set to the empty list unless an initial value is explicitly given. There are no entries that are added or removed from *pending-issues* when the initial value is changed because the initial value is a free-form list.

LISP function used is:

change-initial-value

Delete an Attribute From a Class. An attribute can only be deleted from a class if the class is not a parent of another class. This is so the effect of removing the attribute is minimized within one class and does not cascade into child classes. Since the user of the system is assumed not to be knowledgeable in OORA methods, deleting an attribute from a class will have effects the user will not understand nor may not be able to resolve. Also, keeping an attribute in a class does not cause problems with the model's ability to satisfy the user's requirements.

An attribute cannot be deleted if it is in the *necessary-classes* associative list. This list contains those classes and any of its attributes and services that cannot be removed from the model.

Any entries in *pending-issues* associated with the deleted attribute are removed. Entries are added to *pending-issues* for any attributes and services that reference the deleted attribute.

Entries that can be removed from *pending-issues* are:

(1) (atts-classe class-name attribute-name)

The attribute-name is that of the attribute being deleted.

(2) (atts-atte class-name attribute-name)

The attribute-name is that of the attribute being deleted.

(3) (null-a-set class-name attribute-name)

The attribute-name is that of the attribute being deleted.

Entries that can be added to *pending-issues* are:

(1) (atts-atte class-name attribute-name)

The a-set slot of attribute name uses the deleted attribute.

(2) (check-parameter class-name service-name input-set input-parameter)

The values slot of the input parameter uses the deleted attribute.

(3) (check-parameter class-name service-name input-set output-parameter)

The values slot of the output parameter uses the deleted attribute.

(4) (check-attr-val class-name service-name att-name)

The class-name is the class of the deleted attribute and att-name is the name of the deleted attribute.

LISP functions used are:

delete-attribute

attr-del-check

attr-del-check

atts-classc

atts-attc

check-parameter

check-attr-val

Add an Attribute to a Class. The input is the class name, the new attribute name, a description for the new attribute, and the a-set slot value for the attribute. If the a-set value is not a valid one, the a-set slot is set to null and an entry is added to *pending-issues* indicating the a-set slot needs to be filled in. Entries in *pending-issues* that are resolved by the addition of the attribute are removed from the issues list.

Entries that can be removed from *pending-issues* are:

(1) (atts-attc class-name attribute-name)

The new attribute is used in the a-set of attribute-name.

(2) (check-parameter class-name service-name input-set parameter)

The new attribute is used in the values slot of an input parameter.

(3) (check-parameter class-name service-name output-set parameter)

The new attribute is used in the values slot of an output parameter.

(4) (check-attr-val class-name service-name att-name)

The new attribute is att-name, which is an attribute used in the postcondition as an attribute that has changed as a result of execution of the service.

Entries that can be added to *pending-issues* are:

(1) (null-a-set class-name attribute-name)

The a-set slot of the new attribute is null.

LISP functions used are:

add-attribute

change-attr-a-set

remove-missing-att-entries

change-attr-a-set

atts-attc

check-parameter

check-attr-val

Change the Name of a Service. Changing a service name requires changing the name in the local service name slot and in the message slot of the postconditions of other classes. First, the new service name is checked to ensure it is not the name of an existing service in that class. Service names within a class must be unique.

The name of the service must also be changed wherever it is used in *pending-issues*. Any entries in *pending-issues* that are resolved due to the change of name are removed.

If the name of the service is used in *necessary-classes*, it is changed there.

Entries that can be removed from *pending-issues* are:

(1) (check-messages class-name service-name message)

The message contains the new service name.

LISP functions used are:

change-service-name

change-ser-name-in-messages

change-ser-name-in-pending

remove-missing-serv-entries

Change the Description of Service. Changing the description simply requires replacing the old description with the new. There are no checks to be made, except that the new description must be a string.

LISP functions used are:

change-ser-desc

Change the Input Set of a Service. The input set consists of a parameter name and its type, or legal set of values. The name of a parameter is arbitrary, but it may be used in the precondition or postcondition of a service. The name also must be unique within the input set of a service and must not be the name of an attribute or one of the legal attribute types, such as "enum" and "int". The type of an input parameter must be any of the legal attribute types, a local attribute name, an attribute of another class, a class, or a list consisting of any legal elements.

There are three changes that can be made to one parameter of the input set. Only one parameter of the input set is changed at a time. An existing parameter could be deleted, a parameter could be added, or an existing parameter could be changed.

The input list to any one of these changes is the class name, class-name, the service name, service-name, the existing name and type in a list, old-name-val-list, and the new name and type in a list, new-name-val-list.

(1) Delete an existing parameter.

Since there is no new parameter, the new-name-val-list equals (*delete). The old-name-val-list contains the old name and values, (old-name old-values). First, the input parameter list is searched to ensure the parameter exists. The existing parameter is removed, and then the *pending-issues* list is examined to see if there are any issues relating to the value of the old name. For example, if the old name had a value that contained the name of a class that did not exist, an entry would be added to the pending issues list. Then the precondition and atts slots of the postcondition are examined to see if the deleted parameter name is used. If it is, an entry is added to the *pending-issues* list to indicate the name is no longer valid.

(2) Add a new parameter.

Since there is no old parameter, the old-val-name-list equals (*add). The new val-name-list contains the new name and values, (new-name values). The new-val-name-

list is evaluated to ensure it contains two elements, and the name is legal for an input parameter name. The new parameter is then added to the input set. The values of the new parameter are tested and if they are not legal, an entry is added to *pending-issues*. Also, *pending-issues* is examined to see if any entries can be removed. If there is an entry caused by a deleted input parameter name that was used in the precondition or the postcondition, the name of that missing input parameter is examined. If it is the same as the name of the new parameter, these entries in *pending-issues* can be removed.

(3) Change an existing parameter.

This would require name and values information in both the old-val-name-list and the new-val-name-list. The input set is examined to ensure that the old-val-name-list exists. the new-val-name-list is examined for proper structure and a legal name. There are three possibilities: only the name of the parameter is changed, only the value of the parameter is changed, or both are changed.

If the name is changed and if the old name was used in the precondition or postcondition, the name is changed to the new one. Also, *pending-issues* is examined to see if any entries can be removed. If there is an entry caused by a deleted input parameter name that was used in the precondition or the postcondition, the name of that missing input parameter is examined. If it is the same as the name of the new parameter name, these entries in *pending-issues* can be removed.

If the value is changed, the values of the new parameter are tested and if they are not legal, an entry is added to *pending-issues*. The *pending-issues* is checked to see if there is an entry because of illegal values of the old-val-name-list. If there are, they are removed because the old set of values has been replaced.

Entries that can be removed from *pending-issues* are:

(1) (check-parameter class-name service-name input-set parameter)

The values of a replaced or deleted parameter were not valid.

(2) (:service class-name service-name pre missing input-set para-name)

(3) (:service class-name service-name post missing input-set para-name)

The name of a new parameter matches a missing input parameter name.

Entries that can be added to *pending-issues* are:

(1) (:service class-name service-name pre missing input-set para-name)

(2) (:service class-name service-name post missing input-set para-name)

If an input parameter is deleted or the name changed, and the name is used in the precondition or postcondition, an entry is added stating the parameter name is no longer valid.

(3) (check-parameter class-name service-name input-set parameter)

The values of a new or replaced parameter have invalid references.

LISP functions used are:

change-input-set

unique-para-name

add-to-io-set

in-para-list

replace-name-in-io-set

replace-value-in-io-set

remove-val

add-to-io-set

check-classes-and-atts

check-io-name-pending

replace-value-in-io-set

check-classes-and-atts

unique-para-name

check-classes-and-atts

check-parameter

check-io-name-pending

in-para-list

check-pre-and-post

check-name-pending

Change the Output Set of a Service. The output set of a service consists of a set of output parameters. The name of each the parameters is set to null, and the type or values of an output parameter must be any of the legal attribute types, a local attribute name, an attribute of another class, a class, or a list consisting of any legal element. The name is null because there is no requirement to name the output parameters of a service, just to state its legal values.

There are three changes that can be made to one parameter of the output set. Only one parameter of the output set is changed at a time. An existing parameter could be deleted, a parameter could be added, or an existing parameter could be changed. The inputs required are the class name, the service name, the old values and the new values.

When a parameter is deleted or changed, any entries in pending-issues referring to the old values being illegal are removed. When a parameter is added or changed, the new values are checked to see if they are valid. If they are not, an entry is added to pending-issues.

Entries that can be removed from *pending-issues* are:

(1) (check-parameter class-name service-name output-set parameter)

The values of a replaced or deleted parameter were not valid.

Entries that can be added to *pending-issues* are:

(1) (check-parameter class-name service-name output-set parameter)

The values of a new or replaced parameter have invalid references.

LISP functions used are:

change-output-set

add-to-io-set

remove-val

in-para-list

replace-value-in-io-set

add-to-io-set

check-classes-and-atts

check-io-name-pending

replace-value-in-io-set

check-classes-and-atts

Change the Precondition of a Service. The precondition is a free form list that evaluates to true or false. The new precondition is a list that replaces the old precondition. If the existing precondition is other than null, pending-issues is checked. Any entries referring to an invalid parameter name in the old precondition are removed if the new precondition does not use this parameter.

Entries that can be removed from *pending-issues* are:

(1) (:service class-name service-name pre missing input-set para-name)

The precondition had contained reference to the name of an input parameter that no longer exists. If the precondition is changed and no longer contains reference to the missing name, the entry in *pending-issues* is removed.

LISP functions used are:

change-serv-pre

Change the Atts Slot of the Postcondition of a Service. The atts slot contains a list of attr-val structures that each contain an attribute name and the new attribute value. The changes that can be made is an existing attr-val can be deleted, a new attr-val structure

could be added, or either the attribute name, the attribute value, or both, of an existing attr-val structure can be replaced.

The inputs are the class name (class-name), the service name (service-name), the old attr-val list (old-list), and the new attr-val list (new-list).

(1) A new attr-val structure is added.

The old-list is set to (*add) and the new-list consists of a new attribute and a value in the list form (attribute-name value). The attribute name cannot be the name of any attribute currently in the atts slot, because there should be only one value for an aturibute. The new attribute name is checked to see if it is the name of an attribute currently in the class. If not, the new structure is added but an entry is added to *pending-issues* indicating the attribute does not exist in the class. This allows the user to add the attribute later.

(2) An old attr-val structure is deleted.

The new-list is set to (*delete) and the old-list consists of an existing attribute and value in list form, (attribute-name value). The currents atts slot is examined to ensure the old-list exists. If so, the old-list is removed from the atts slot. Any entries in pending-issues referring to an invalid attribute name in the old structure are removed. Also, any entries in *pending-issues* referring to the use of a non-existent input parameter name in the value portion are removed if the name is not used in the value slot of any of the other structures in atts.

(3) An old attr-val structure is changed.

Either the attribute name, the value, or both can be changed. When the name is changed, the new name is checked to ensure it is not the name of an existing structure in atts. If it isn't, the name is changed, and then checked to determine if it is the name of an attribute in the class. If it is not, an entry is added to pending-issues indicating the attribute-value structure refers to a non-existent attribute. When the value is changed, any

entries in *pending-issues* referring to the use of a non-existent input parameter name in the old value are checked. These are removed if the name is not used in the new value or used in any of the other value slots in atts.

Entries that can be removed from *pending-issues* are:

(1) (check-attr-val class-name service-name att-name)

An old attribute name, that is now replaced or deleted, was invalid.

(2) (:service class-name service-name post missing input-set para-name)

The value is changed or deleted, and there is an entry about an invalid input parameter, and that invalid input parameter is not used in the new value.

Entries that can be added to *pending-issues* are:

(1) (check-attr-val class-name service-name att-name)

The attribute name of a new attr-val (either new or replaced) is not the name of an attribute in the class.

LISP functions used are:

change-serv-post-atts

- unique-attr-atts
- add-to-postf-atts
- in-post-atts
- remove-from-post-atts
- replace-post-atts-attr
- replace-post-atts-value

add-to-postf-atts

- check-post-atts-attr

remove-from-post-atts

- check-post-atts-attr
- check-post-atts-missing-input-para

replace-post-atts-attr

check-post-atts-attr

replace-post-atts-value

check-post-atts-missing-input-para

check-post-atts-attr

check-attr-val

Change the Messages Slot of the Postcondition of a Service. The messages slot consists of a list of messages. Each message is a list consisting of a class name and a service name. A message can be deleted, a message added, or a message replaced. The input is the class name, the service name, the old message and the new message.

If a message is added or replaced, the new message is checked for the validity of the class and service. If either is invalid, an entry is added to *pending-issues*. When an existing message is deleted or replaced, entries in pending-issues on the invalidity of the old messages class and/or service are removed. When a new message is added, it must not be the same as an existing message.

Entries that can be deleted from *pending-issues* are:

(1) (check-messages class-name service-name message)

If a message with an entry is changed or deleted.

Entries that can be added to *pending-issues* are:

(1) (check-messages class-name service-name message)

A new message has invalid references.

LISP functions used are:

change-serv-post-mess

unique-message

add-to-post-messages

remove-from-post-messages

replace-post-message

add-to-post-messages

check-post-atts-mess

remove-from-post-messages

check-post-atts-mess

replace-post-message

check-post-atts-mess

Delete a Service. The service of a class that is a parent cannot be deleted for the same reason as that for not deleting an attribute of a class that is a parent. If the service can be deleted, all entries in *pending-issues* for that service are deleted. Deleting the service may cause messages in other services to have invalid references, which causes entries to be added to *pending-issues*.

A service cannot be deleted if it is in the *necessary-classes* list.

*Entries that can be deleted from *pending-issues* are:*

(1) (:service class-name service-name pre missing input-set parameter)

(2) (:service class-name service-name post missing input-set parameter)

(3) (check-parameter class-name service-name input-set parameter)

(4) (check-parameter class-name service-name output-set parameter)

(5) (check-attr-val class-name service-name att-name)

(6) (check-messages class-name service-name message)

All entries with service-name the same as the deleted service are removed.

Entries that can be added to *pending-issues* are:

(1) (check-messages class-name service-name message)

Messages that have invalid references due to the deletion of the service.

LISP functions used are:

delete-service

remove-serv-entries

serv-del-check

Add a Service. There are two methods for adding services. The first adds any generic service without using a service template. The second uses one of four service templates that are set up for services that either get or change the value of an attribute.

The method without using a template takes the class name, the new service name and the description of the service and creates a service with the slots for input-set, output-set, precondition and postcondition set to null. The function add-to-io-set would be used to enter input and output parameters, change-serv-pre for the pre, change-serv-post-atts for the atts portion of the postcondition, and change-serv-post-mess for the messages portion of the postcondition. Any entries in *pending-issues* resolved by the addition of the service are removed.

Adding a service based on a template requires input of the class name, a new service name, a template name, an attribute name and an optional message list. The four templates supported are changing the value of an attribute, returning the value of an attribute, adding a value to an attribute that is a list of values, and removing a value from an attribute that is a list of attributes. The service is then completely created from that template, including input and output sets, the precondition and the postcondition. Any entries in *pending-issues* resolved by the addition of the service are removed.

Entries that can be deleted from *pending-issues* are:

(1) (check-messages class-name service-name message)

The new service name is used in the message.

LISP functions used are:

add-service

remove-missing-serv-entries

add-template

remove-missing-serv-entries

Change the Whole/Part or Relation Structure of a Class. The whole/part and relation slots in a class are made up of a list of individual relations. A single relation structure that is part of the whole-part and relation slots in a class can change in one of five ways:

(1) The ranges in the relation can change. Two new ranges are used as a replacement for the existing ranges. This requires that the relation exists in at least one class and the two ranges are lists of two elements. It is possible that the other class of the relation does not exist. This can occur when a relation is added or changed and one class of the relation is not in the model. When this occurs, an entry is added to *pending-issues* indicating there is a missing class and an associated relation. If the other class exists, the ranges are changed in the other class as well. If the other class does not exist, there must already be an entry in *pending-issues* on the nonexistence of that class and the relation. Therefore, the relation in *pending-issues* is updated to reflect the new range information.

(2) A relation is added. An existing class and a new relation is input. The new relation is added to the class if it does not already exist in the class and the relation is of proper form. The relation is of proper form if the two classes in the relation are different, one of the classes in the relation is the existing class to which the relation is to be added, and the ranges are lists of two elements. If the other class in the relation exists in the model, the relation is added to that class. If the other class does not exist, an entry is added to *pending-issues* with the class name and the added relation.

(3) A relation is deleted. The input is an existing class and the relation to be deleted. The relation must be in the existing class. The relation is deleted in the existing class. If the other class in the relation exists, the relation is deleted in that class as well. If the

other class does not exist, the entry in *pending-issues* for that class and relation is deleted.

(4) The other class of a relation is changed. The input is an existing class, a relation of that class, and a new class that replaces the current other class of the relation. The relation must exist in the input parameter class. If the current other class of the relation exists in the model, the relation is removed from that class. If it does not, the entry in pending issues for the other class and the relation is deleted. The new class replaces the existing other class in the relation. If the new class exists in the model, the relation is added to it. If the new class does not exist, and entry is added to *pending-issues*.

(5) The name of the relation is changed. This can only occur in other general relations, and not whole/part relations. The input is an existing class, a relation in that class, and a new relation name. The relation name is changed in that class. If the other class in the relation exists, the name is changed in that class as well. If it does not exist, the entry in *pending-issues* on the class and relation is changed to reflect the new relation name.

Entries that can be deleted from *pending-issues* are:

(1) (missing-class-and-relation class-name the-relation)

When a relation is deleted and the old relation had a missing class.

Entries that can be added to *pending-issues* are:

(1) (missing-class-and-relation class-name the-relation)

The other class in the relation does not exist in the model. Used when a relation is added or modified.

LISP functions used are:

change-relation-name

remove-relation-class-missing

add-relation-class-missing

add-new-relation

    add-relation-class-missing

delete-relation

    remove-relation-class-missing

change-relation-class

    remove-relation-class-missing

    add-relation-class-missing

change-relation-name

    remove-relation-class-missing

    add-relation-class-missing

remove-relation-class-missing

add-relation-class-missing

Change the Parents of a Class. There are three permissible changes to the parents of a class. *Changes in the parents of a class are changes made to the inheritance slot of a class.*

(1) Remove a parent of a class. This requires that the class must not be the parent of another class. If the class is the parent of other classes, a change in the parent would affect all classes that inherit from it. This is a far-reaching change that changes the basic structure of the model. Since it is assumed the user is not familiar with object-oriented techniques, this type of dramatic change is not permitted. If the user finds it necessary to change the structure to this extent, the domain model should be redone to more accurately reflect the domain. If the class is not a parent of another class, the desired parent is removed. The attributes and services of the class are then checked to determine if there are attributes used that are no longer valid because the attributes of the removed parent are no longer available. If there are any attributes or services that are invalid, the information is added to the *pending-issues* list.

(2) Add a parent to a class. This requires that the new parent exist in the model. The parent is added, and the *pending-issues* list examined for any issues about non-valid attributes and services that can be removed.

(3) Change a parent of a class. This requires the removal of the old parent and the addition of a new parent, so all the requirements of the removal and addition of a parent apply.

Entries that can be deleted from *pending-issues* are:

(1) (atts-attc class-name att-name)

Used when a parent is removed which causes an attribute to become invalid.

(2) (check-parameter class-name service-name input-set parameter)

(check-parameter class-name service-name output-set parameter)

Used when a parent is removed which causes the value slot of an input or output parameter to become invalid.

(3) (check-attr-val class-name service-name att-name)

Used when a parent is removed which causes the attribute to become invalid.

Entries that can be added to *pending-issues* are:

(1) (atts-attc class-name att-name)

Removed when a parent is added which causes an attribute to become valid.

(2) (check-parameter class-name service-name input-set parameter)

(3) (check-parameter class-name service-name output-set parameter)

Removed when a parent is added which causes the value slot of an input or output parameter to become valid.

(4) (check-attr-val class-name service-name att-name)

Removed when a parent is added which causes the attribute to become valid.

LISP functions used are:

remove-parent

atts-attc

check-parameter

check-attr-val

add-parent

atts-attc

check-parameter

check-attr-val

change-parent

remove-parent

add-parent

Add a Class. The inputs are the new class name and a class description. The remainder of the class's slots are set to null. The attributes are added using add-attribute. The services are added using either add-service or add-template. The parents are added using add-parent, and the relations are added using add-new-relation.

Any entries in *pending-issues* resolved by adding the class are removed. Also, if there are relations for that class in *pending-issues*, they are added to the class and removed from *pending-issues*.

Entries that can be deleted from *pending-issues* are:

(1) (atts-classc class-name att-name)

(2) (atts-attc class-name att-name)

An attribute contains the new class name in the a-set slot.

(1) (check-parameter class-name service-name input-set parameter)

(2) (check-parameter class-name service-name input-set parameter)

The values slot of an input or output parameter contains the new class.

(3) (check-messages class-name service-name message)

The message slot contains the new class.

LISP functions used are:

add-class

new-class-pending

add-rel-to-new-class

Delete a Class. A class can only be deleted if it is not the parent of another class. This follows the same reasoning as that for deleting an attribute or service of a parent class. Also, a class cannot be deleted if it is in the *necessary-classes* list. Any relations in the class are removed from the other class in the relation, if it exists in the model. Deleting a class causes all entries in *pending-issues* for that class to be deleted, and entries in *pending-issues* added for all attribute and service slots that now have invalid references to the deleted class.

Entries that can be deleted from *pending-issues* are:

(1) (atts-classc class-name att-name)

(2) (atts-attc class-name att-name)

(3) (:service class-name service-name pre missing input-set para-name)

(4) (:service class-name service-name post missing input-set para-name)

(5) (check-parameter class-name service-name input-set parameter)

(6) (check-parameter class-name service-name input-set parameter)

(7) (check-attr-val class-name service-name att-name)

(8) (check-messages class-name service-name message)

(9) (null-a-set class-name attribute-name)

If the deleted class name equals class-name, the entries are removed.

(10) (missing-class-and-relation class-name the-relation)

If the deleted class name is the other class (not class-name) in the relation, the entry is removed.

Entries that can be added to *pending-issues* are:

(1) (atts-classc class-name att-name)

The deleted class is used in the a-set slot of an attribute.

(2) (atts-attc class-name att-name)

The deleted class contained an attribute that is used in the a-set slot of an attribute.

(3) (check-parameter class-name service-name input-set parameter)

The deleted class or one of its attributes is used in the values slot of the input set of a service.

(4) (check-parameter class-name service-name output-set parameter)

The deleted class or one of its attributes is used in the values slot of the output set of a service.

(5) (check-messages class-name service-name message)

A service of the deleted class is used in the messages slot of the postcondition of a service.

LISP functions used are:

delete-class

delete-relation

remove-a-class-pending

attr-del-check

serv-del-check

Verify a Class. The class verification is used to require the user to review all classes, to include their attributes and services, that were part of the original model and kept in the revised model. Each class, as well as each attribute and service, has a verify slot that is initially set to "false", indicating the user has not verified the need for that class or component yet. The user must set all verify slots to true before the problem model can be considered complete. The verify slot of a class cannot be set to true until the verify slots of all the attributes and services in that class are set to true.

When a new class, attribute or service is created, the verify slot is initially set to true. This is because the user has obviously determined there is a need for that component.

As stated above, the model is not considered complete until the verify slot of all classes is set to true. Therefore, when the model is first created, there is already an entry in *pending-issues* stating that all the classes have not been verified. This entry is removed when all the classes have been verified.

Entry that can be deleted from *pending-issues*:

(1) (classes need verified)

This is deleted when all classes in the model have been verified.

b. LISP functions used:

(verify-class class-name)

Pending Issues. The list of pending issues is critical to the development of the problem model. The requirement for resolution of entries on the list insures the problem model is consistent and complete with respect to all guidelines and rules. The problem model could be developed without the use of this list and still be kept consistent and complete. However, this would require much more intellectual work on the part of the user and it would be far more error prone. The user would have make changes in certain, strict orders. For example, in order to delete a class without using a pending issues list, all references to that class in other classes would have to be changed first, before the class is deleted. The use of the pending issues list is much more in keeping with the practice of good software engineering by providing as much support as possible for the process which was most intuitive for the user.

The following are all possible entries in the *pending-issues* list, and when they are added and removed from the list.

1. A class name is not valid in the a-set slot of an attribute.

   (atts-classc class-name att-name)

a. Added when:

The a-set slot of an attribute is changed and a class in the a-set is not valid.

A class is deleted that is used in the a-set slot of an attribute.

b. Removed when:

The a-set slot of an attribute is changed, the old a-set had an entry in pending issues, and the new a-set is valid.

The attribute whose a-set slot was invalid is deleted.

The missing class in the a-set slot is added.

The class of the attribute whose a-set slot was invalid is deleted.

2. An attribute name is not valid in the a-set slot of an attribute.

(atts-attc class-name att-name)

a. Added when:

The a-set slot of an attribute is changed and a class and/or attribute in the new a-set is not valid.

When removing a parent from a class, an attribute used in the a-set slot of an attribute becomes invalid.

A class is deleted which contains an attribute used in the a-set slot of an attribute.

An attribute is a class is deleted that is used in the a-set of other attributes.

b. Removed when:

The a-set slot of an attribute is changed, the old a-set had an entry in pending issues, and the new a-set is valid.

The attribute which had an invalid a-set slot is deleted.

The missing attribute of the a-set slot is added.

A parent is added to a class which contains the missing attribute for an a-set slot.

A class is added which contains the missing attribute of the a-set slot.

A class is deleted which contains the attribute with the invalid a-set slot.

3. An input-parameter name is used in the precondition no longer exists.

(:service class-name service-name pre missing input-set para-name)

a. Added when:

An input parameter is deleted that contains a name used in the precondition.

b. Removed when:

A parameter is added whose name is that of the missing input parameter.

The name of a parameter is changed and the new name matches the missing parameter name.

The precondition is replaced with one that does not use the missing input parameter name.

The service is deleted that contains the precondition.

A class is deleted that contains the service with the precondition.

4. An input parameter name used in the atts slot of the postcondition no longer exists.

(:service class-name service-name post missing input-set para-name)

a. Added when:

An input parameter is deleted that is used in the atts slot of a postcondition.

b. Removed when:

The name of a parameter is changed and the new name matches the missing parameter name.

An attribute-value is deleted that contains the non-existent input parameter name.

The value part of an attribute-value is changed to remove the non-existent input parameter name.

A service is deleted that contains the postcondition with the non-existent input parameter name.

A class is deleted that contains the service with the postcondition with the non-existent input parameter name.

5. A class and/or local or external attribute of the value slot of an input set is invalid.

(check-parameter class-name service-name input-set parameter)

a. Added when:

An attribute is deleted that is used in the input parameter of a service.

When removing a parent from a class, an attribute used in the values slot of an input parameter of a service becomes invalid.

The value of a parameter is changed and the new value is invalid.

A parameter is added whose value slot is invalid.

A class is deleted that is used in the value slot of an input parameter.

b. Removed when:

An attribute is added that satisfies the value slot of an input parameter.

The invalid parameter is deleted.

The value slot of a parameter is changed to a valid value.

A service is deleted which contains the input parameter with the invalid value slot.

A parent is added to a class which contains the missing attribute needed to satisfy the value slot of an input parameter.

A class is added that satisfies the value slot of an input parameter.

A class is deleted that contains a service with an invalid values slot for an input parameter.

6. A class and/or local or external attribute of the value slot of the value slot of an output set is invalid.

(check-parameter class-name service-name output-set parameter)

a. Added when:

An attribute is deleted that is used in the input parameter of a service.

When removing a parent from a class, an attribute used in the values slot of an output parameter of a service becomes invalid.

The value slot is changed and the new value is invalid.

A parameter is added whose value slot is invalid.

A class is deleted that is used in the value slot of an output parameter.

b. Removed when:

An attribute is added that satisfies the value slot of an input parameter.

The invalid output parameter is deleted.

The value slot of a output parameter is changed to a valid value.

A service is deleted which contains the output parameter with the invalid value slot.

A parent is added to a class which contains the missing attribute needed to satisfy the value slot of an output parameter.

A class is added that satisfies the value slot of an output parameter.

A class is deleted that contains a service with an invalid values slot for an output parameter.

7. Invalid attribute name in the atts slot of the postcondition.

(check-attr-val class-name service-name att-name)

a. Added when:

An attribute is deleted that is used in the input parameter of a service.

An attribute-value is added containing an invalid attribute name.

The attribute name of an attribute-value is changed to an invalid attribute name.

When removing a parent from a class, an attribute used in the attr-val slot of the postcondition slot becomes invalid.

b. Removed when:

An attribute-value is deleted that contains an invalid attribute name.

The attribute name of an attribute-value is changed from an invalid name to a valid name.

A service is deleted that contains a postcondition with an invalid attribute name.

A parent is added to a class which contains the missing attribute that satisfies the invalid attribute name.

A class is deleted which contains the service with the reference to the invalid attribute name.

8. Invalid class name and/or service name in a message of the messages slot of the postcondition.

(check-messages class-name service-name message)

a. Added when:

A message is added with an invalid class or service name.

A message is replaced with one with an invalid class or service name.

A service is deleted that is used in the messages of another service.

A class is deleted which contains services used in the messages of other services.

b. Removed when:

A message is deleted that contains reference to invalid classes or services.

A invalid message is replaced.

A service is deleted which contains invalid messages.

A service is added which satisfies the invalid messages in another service.

A class is added with services that satisfy the invalid messages.

A class is deleted which contains services with invalid messages.

9. A class in a relation (either whole/part or other relation) is not in the model.

(missing-class-and-relation class-name the-relation)

a. Added when:

A relation is added using a non-existent class.

The other class of a relation is changed to a non-existent class.

b. Removed when:

A relation is deleted which contains reference to a non-existent class.

The other class of a relation is changed from a non-existent class to an existing class.

A class is deleted that contains a relation with reference to a non-existent class.

The non-existent class in a relation is added.

10. The value of the a-set in an attribute is null.

    (null-a-set class-name attribute-name)

    a. Added when:

    An attribute is added with a null a-set.

    b. Removed when:

    An attribute with a null a-set is deleted.

    A class is deleted that contains an attribute with a null a-set.

11. All classes in the model have not been verified.

    a. Added when:

    The model is first created.

    b. Removed when:

    All classes in the model have been verified.

Advisory Issues. These entries are created and shown to the user on the request of the user. These are also used by the developer of the domain model to evaluate the model. The list is recreated each time it is requested. These issues are advisory only, and are not required to be resolved before the model is complete.

1. The class is not connected to any other class in the model.

   (connectionp class-name)

2. The class is a parent and it has only one subclass.

   (two-subclass-check class-name)

3. The class has zero or one attributes.

   (one-attributep class-name)

4. The class has zero or one service.

   (one-servicep class-name)

5. The class shares 80% of its attributes and services with another class.

   (share-att-serv class-name)

6. The depth of a class in the inheritance structure is greater than 2.

   (class-depth class-name)

This section discussed and analyzed the LISP functions used to modify the problem model. The modification of the problem model is a complicated process that requires an analysis of each change to evaluate any possible inconsistencies and incompleteness caused by each change. These are handled in one of three ways. In some cases the change is not allowed. In a majority of the cases, the change is allowed but the inconsistencies and incompleteness are recorded in the pending issues list for later resolution. In some cases, the system cannot determine if the problem is one that must be resolved, so the issues is placed on the advisory issues list for possible resolution by the user. The use of a pending issues list is an important one for the OAKS system. It allows many model changes to be made without imposing any order on the changes, yet ensures that the model will be consistent and complete when the pending issues are resolved.

The last section of this chapter discusses the user interface implemented for OAKS.

User Interface

Background. The functions shown in the model modification portion can be used to fully manipulate the model and create a new problem. However, this would force the user of OAKS to type in LISP commands at the LISP prompt. Although the object of this research is a proof-of-concept system rather than a production system, a more appropriate interface for the expected user can better illustrate the concept of an automated system for

OORA. Hence, a window-based user interface (UI) was created using LISPView. LISPView was chosen because it was a package available in the SUN Common LISP environment that contained all the features required for a windowed user interface. LISPView uses CLOS classes for each of its components, such as windows and menus.

Because OAKS is a proof-of-concept system, extensive user input checking was not implemented. The OAKS system, for the most part, requires the user input to be in the expected form. For example, if a list is expected, the OAKS system requires a list to be input. the checking that is done is based on the desired structure of the changed model. For example, a new class is not allowed to be created if it has the same name of an existing class in the model.

The following is a guide to using the OAKS LISPView user interface. It also explains the connection between the user interface and the OAKS domain model and model modification functions.

Overview of the UI. *Figure 7-2 is a drawing of the windows and menus available in* OAKS.

There are three main windows. The window labeled 1 displays either the entire model or the components of one class depending on which class has been selected by the user. The window labeled 2 displays the component that is currently selected by the user. The window labeled 3 always displays the entries in the *pending-issues* list. What is displayed in each window is based on the selection the user makes through the use of the menus that run across the top bar. These menus and the actions generated by the selections on the menus are described in the next sections.

To see what item is currently selected on the menu, the menu button is pressed using the left mouse button. To select a different menu item, the right mouse button is used to produce a pull-down list of menu items. In some cases, there are submenus to these pull-down menus.

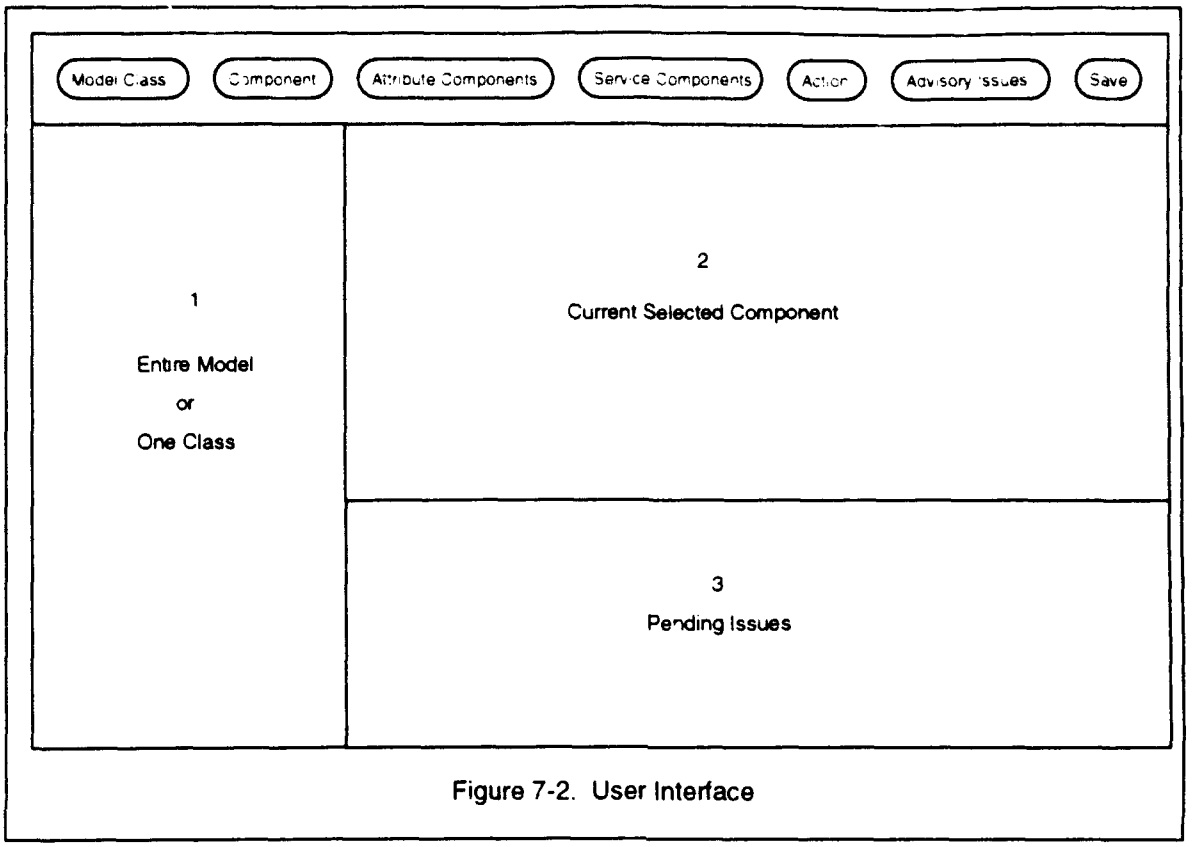


Figure 7-2. User Interface

The general process a user would go through in using OAKS is to first initiate a SUN Common LISP environment containing the LISPView and CLOS packages. At the LISP prompt, the user would type (load "oaks.lisp"). This file loads the files "oaksd.lisp", which contains the domain model structure and the domain model, "oaksno.lisp" which contains the model evaluation functions, "oaksmod.lisp" which contains the model modification functions, "oaksave.lisp" which saves the changed model to a file, executes the function "read-data", which reads from the file "userfil", and then loads "oaksui.lisp" which contains the LISPView user interface. The file "userfil" contains the problem model. This is the model the user modifies to create a model for the particular problem of interest. When OAKS is first used, "userfil" contains the unmodified domain model. The user then

types (in-package 'oaks) at the LISP prompt. All the files are loaded into this package. The next step is to select the component the user would like to view or modify using the "Component" menu, and then use the "Action" menu to change that component. The "Action" menu may bring up a pop-up box that gathers the user input. User feedback comes in the form of another box that tells the user of the consequences of an action or errors in entering information. As changes are made, the entries in window 3, pending issues, are changed to reflect the effect of those changes. The model is not complete until there are no longer any entries in pending issues. Appendix B walks through an example OAKS session.

To create the file "userfil" for the first session with OAKS requires that 'userfil" contains the domain model with no changes. This is done by loading "oaksd.lisp", "oaksno.lisp", "oaksmod.lisp" and "oaksave.lisp". Then the function "write-data" is run. Because the only model loaded is the domain model, the domain model will be saved to "userfil".

Model/Class Menu. The "Model/Class" menu selects either the entire model or one class in the model. The pull-down menu produces a list with one menu selection of "Entire Model" and the remaining menu selections are the names of the classes in the model. The menu will change as the classes in the model are added, deleted, or the names changed.

If "Entire Model" is selected, window 1 will show a list of every class in the model with any parents of a class shown indented under the class. Window 2 will be blank. The "Component" menu is inactive, which means it cannot be used. The "Attribute Components" and "Service Components" menus are also inactive. The "Action" menu consists of the action "Add a Class", which is the only action allowed on the entire model.

If one class is selected, a class and its components will be shown in window 1. The components shown are the class name, description, a list of the attribute names but not the

attribute components, a list of the service names but not the service components, the whole/part relations, the other relations, the parents of the class, and if the class is verified or not. The "Component" menu is active. The "Attribute Components" and "Service Components" menus are inactive. The "Action" menu consists of the actions "Delete the Class", "Verify the Class", "Add an Attribute", "Add a Service", and "Add Service Using Template". When a class is initially selected, no particular component of the class is selected, but the entire class is selected. Therefore, window 2 is blank. Once a class is selected, any component of the class can be selected using the "Component" menu. When a component is selected, window 2 will show that component.

At any point in the development of the problem model, if a new class is selected using the "Model/Class" menu, any components selected for the previous class are cleared and the entire new class is selected, with window 2 blank, the "Component" menu active, and the "Attribute Components" and "Service Components" menu inactive. The "Action" menu once again consists of the actions "Delete the Class", "Verify the Class", "Add an Attribute", "Add a Service", and "Add Service Using Template".

The Component Menu. The "Component" menu is active whenever a class is selected, rather than the entire model. This menu is used to select a particular component of a class. The menu consists of "Entire Class", "Class Name", "Class Description", "One Attribute", "One Service", "Whole-Part", "Relations", and "Inheritance". The "One Attribute" and "One Service" menu selections display submenus consisting of the attribute and service names, respectively.

The "Entire Class" selection is the initial selection when a class is first chosen from the "Model/Class" menu. Window 2 is blank and the "Action" menu consists of the actions "Delete the Class", "Verify the Class", "Add an Attribute", "Add a Service", and "Add Service Using Template".

If "Class Name" is selected, the class name is shown in window 2. The "Action" menu consists of "Change Class Name".

If "Class Description" is selected, the class description is shown in window 2. The "Action" menu consists of "Change Class Description".

If "One Attribute" is selected, a submenu is shown of all the attribute names. This submenu changes when attributes are added, deleted, or the names are changed. Once an attribute name is selected, the "Attribute Components" menu is activated. The attribute components are shown in window 2. These components are the attribute name, description, the legal values, and whether or not the attribute has been verified. When an attribute is first selected, the entire attribute is selected. Therefore, the "Action" menu contains "Delete the Attribute", and "Verify the Attribute". To select a particular attribute component, the "Attribute Components" menu is used. The "Attribute Components" menu is only active while an attribute is selected. Once any other component of a class is selected, the "Attribute Components" menu is inactive.

If "One Service" is selected, a submenu is shown of all the service names. This submenu changes when services are added, deleted, or the names are changed. Once a service name is selected, the "Service Components" menu is activated. The service components are shown in window 2. These components are the service name, description, input set, output set, precondition, postcondition, attributes changed as a result of the service, and messages. When a service is first selected, the entire service is selected. Therefore, the "Action" menu contains "Delete the Service", and "Verify the Service". To select a particular service component, the "Service Components" menu is used. The "Service Components" menu is only active while a service is selected. Once any other component of a class is selected, the "Service Components" menu is inactive.

If "Whole-Part" is selected, the whole-part relations of the class are shown in window 2. The "Action" menu consists of "Add Whole/Part Relation", "Remove Existing Whole/Part Relation", "Change Ranges", and "Change Other Class".

If "Relations" is selected, the other general relations of the class are shown in window 2. The "Action" menu consists of "Add an Other Relation", "Remove an Other Relation", "Change Ranges", "Change Other Class", and Change Relation Name".

If "Inheritance" is selected, the parents of the class are shown in window 2. The "Action" menu consists of "Add a Parent", "Remove a Parent", and "Change Existing Parent".

The Attribute Components Menu. The "Attribute Components" menu is active when an attribute of a class is selected. The menu consists of "Entire Attribute", "Name", Description", "Initial Value" and "Legal Values".

When an attribute is first selected, the entire attribute is selected, which is the same as selecting "Entire Attribute". The attribute components are shown in window 2. These components are the attribute name, description, the legal values, and whether or not the attribute has been verified. The "Action" menu contains "Delete the Attribute", and "Verify the Attribute". Window 2 will not change based on the component of the attribute selected. The window will always show all components of the attribute as long as the attribute is selected. The selection of items on the attribute component menu will affect the choices available on the "Action" menu.

If "Name" is selected, the "Action" menu consists of "Change Attribute Name".

If "Description" is selected, the "Action" menu consists of "Change Attribute Description".

If "Initial Value" is selected, the "Action" menu consists of "Change Initial Value".

If "Legal Values" is selected, the "Action" menu consists of "Change Legal Values".

The Service Components Menu. The "Service Components" menu is active when a service of a class is selected. The menu consists of "Entire Service", "Name", "Description", "Input Set", "Output Set", "Precondition", "Postcondition Attributes", and "Postcondition Messages".

When a service is first selected, the entire service is selected, which is the same as selecting "Entire Service". The service components are shown in window 2. These components are the service name, description, input set, output set, precondition, postcondition, attributes changed as a result of the service, and messages. The "Action" menu contains "Delete the Service", and "Verify the Service". Window 2 will not change based on the component of the service selected. The window will always show all components of the service as long as the service is selected. The selection of items on the "Service Components" menu will affect the choices available on the "Action" menu.

If "Name" is selected, the "Action" menu consists of "Change Service Name".

If "Description" is selected, the "Action" menu consists of "Change Service Description".

If "Input Set" is selected, the "Action" menu consists of "Add Input Parameter", "Remove Existing Input Parameter", and "Change Existing Input Parameter".

If "Output Set" is selected, the "Action" menu consists of "Add Output Parameter", "Remove Existing Output Parameter", and "Change Existing Output Parameter".

If "Precondition" is selected, the "Action" menu consists of "Change Precondition".

If "Postcondition Attributes" is selected, the "Action" menu consists of "Add an Attribute/Value", "Remove Existing Attribute/Value", and "Change Existing Attribute/Value".

If "Postcondition Messages" is selected, the "Action" menu consists of "Add Message to Postcondition", "Remove Message From Postcondition", and "Change Existing Message in Postcondition".

The Action Menu The "Action" menu is used to make all modification to the model. The entries on the "Action" menu change to reflect the component that is currently selected. The following will go through each possible entry in the "Action" menu and the actions that are taken if that entry is selected. There are two pop-up boxes that are shown as the result of selecting an action. One is a pop-up data collection box that collects user input, if any is required. The user enters what is requested and when the user is done, selects the "Done" button on the bottom of the box. The second is a pop-up message box that tells the user any problems with the requested action, such as it could not be taken because some input was invalid, or that there was a pending issue entry created as a result of the change. The box contains a push pin in the upper left hand corner. When the push pin is selected, it is "pulled out" and the message box disappears. After the action is taken, all the windows are refreshed, and they will reflect any changes in the current components and the pending issues.

One peculiarity of the system is the ":", before the "c" or "a" used in values slot of the input and output sets of services, to indicate it is an external class or attribute respectively, is not shown when that component is shown in a window. The user must know it is always there and also know to insert it when entering a new component. For example, if the values slot of an input set is "(:c aircraft)", indicating the parameter is an object of the class aircraft, the ":" will not be shown in window 2 when the service is selected. It will be shown as "(c aircraft)". Also, if the user wants to change the input set values to another class, such as "aircrew", the user would have to enter "(:c aircrew)". This peculiarity is due to the way LISP handles names that are preceded by a colon. They are treated as special keywords.

Each of the possible entries in the "Action" menu will gather any data necessary to carry out that action using the data collection pop-up box. After any necessary data is collected, a LISP function is called to carry out that action. This function is described in

the previous section that describes the OAKS model modification functions. Any message to the user, such as "The class name is already the name of a class in the model", is shown using the pop-up message box. The following summarizes the data collected for each possible action in the "Action" menu item and the LISP function used to carry out that action.

"Add a Class"

Information collected:	Class name
	Class description
LISP function called:	add-class

"Delete a Class"

Information collected:	None (use the currently selected class)
LISP function called:	delete-class

"Change Class Name"

Information collected:	New class name
LISP function called:	change-class-name

"Change Class Description"

Information collected:	New description
LISP Function called:	change-class-desc

"Verify the Class"

Information collected:	None (just set to verified)
LISP function called:	None - set slot to true

"Add an Attribute"

Information collected:	Name
	Description
	Base value
	Lower value (optional)
	Upper value (optional)
LISP function called:	add-attribute

"Delete the Attribute"

Information collected:	None (delete the selected attribute)
LISP function called:	delete-attribute

"Change Attribute Name"

Information collected:	New name
LISP function called:	change-att-name

"Change Attribute Description"

Information collected:	New description
LISP function called:	change-att-desc

"Change Attribute Legal Value"

Information collected:	Base value
	Lower value (optional)
	Upper value (optional)
LISP function called:	change-attr-a-set

"Verify the Attribute"

Information collected: None

LISP function called: None (set slot to true)

"Add a Service"

Information collected: New service name

New service desc

LISP function called: add-service

"Add Service Using Template"

Information collected: Template name (change, return, add, remove)

Attribute name

Service name

LISP function called: add-template

"Delete the Service"

Information collected: None (delete current selected service)

LISP function called: delete-service

"Change Service Name"

Information collected: New name

LISP function called: change-service-name

"Change Service Description"

Information collected: New description

LISP function called: change-serv-desc

"Add Input Parameter"

Information collected: New parameter name

New parameter values

LISP function called: change-input-set

"Remove Existing Input Parameter"

Information collected: Parameter name

Parameter values

LISP function called: change-input-set

"Change Existing Input Parameter"

Information collected: Old parameter name

Old parameter value

New parameter name

New parameter value

LISP function called: change-input-set

"Add Output Parameter"

Information collected: New parameter value

LISP function called: change-output-set

"Remove Existing Output Parameter"

Information collected: Old parameter values

LISP function called: change-output-set

"Change Existing Output Parameter"

Information collected:	Old parameter values
	New parameter values
LISP function called:	change-output-set

"Change Precondition"

Information collected:	New precondition
LISP function called:	change-serv-pre

"Add an Attribute/Value"

Information collected:	New attribute name
	New attribute value
LISP function called:	change-serv-post-atts

"Remove an Existing Attribute/Value"

Information collected:	Old attribute name
	Old attribute value
LISP function called:	change-serv-post-atts

"Change Existing Attribute/Value"

Information collected:	Old attribute name
	Old attribute value
	New attribute name
	New attribute value
LISP function called:	change-serv-post-atts

"Add Message to Postcondition"

Information collected:	Class name
	Service name
LISP function called:	change-serv-post-mess

"Remove Message From Postcondition"

Information collected:	Class name
	Service name
LISP function called:	change-serv-post-mess

"Change Existing Message in Postcondition"

Information collected:	Old class name
	Old service name
	New class name
	New service name
LISP function called:	change-serv-post-mess

"Verify the Service"

Information collected:	None
LISP function called:	None (set the slot to true)

"Add Whole/Part Relation"

Information collected:	Class1
	Range1
	Class2

Range2

LISP function called: add-new-relation

"Remove Existing Whole/Part Relation"

Information collected: Class1

Range1

Class2

Range2

LISP function called: delete-relation

"Change Ranges"

Information collected: Old relation name

Class1

Range1

Class2

Range2

New range1

New range2

LISP function called: change-relation-range

"Change Other Class"

Information collected: Relation name

Class1

Range1

Class2

Range2

New other class

LISP function called: change-relation-class

"Add an Other Relation"

Information collected: Relation name

Class1

Range1

Class2

Range2

LISP function called: add-new-relation

"Remove an Other Relation"

Information collected: Relation name

Class1

Range1

Class2

Range2

LISP function called: delete-relation

"Change Relation Name"

Information collected: Relation name

Class1

Range1

Class2

Range2

New relation name

LISP function called: change-relation-name

"Add a Parent"

Information collected: Parent to be added

LISP function called: add-parent

"Remove a Parent"

Information collected: Parent to be removed

LISP function called: remove-parent

"Change Existing Parent"

Information collected: Parent to be changed

LISP function called: change-parent

Advisory Issues Button. The "Advisory Issues" button puts a list of the advisory issues in window 2 when it is pushed. The advisory issues are removed from window 2 whenever any component of the model is selected through one of the other menus.

Save Button. The "Save" button saves the current state of the problem model in a file called "userfil". Each time OAKS is used the file "userfil" is used to create the problem model. The first time OAKS is used, the problem model is the same as the domain model. When changes are made to the domain model, the "Save" button saves these changes in "userfil" so they are present in any subsequent sessions.

This chapter described the development and analysis of the OAKS system. This started with the structure of the domain model. Guidelines and rules based on the structure of the domain model were developed and implemented. Next, the guidelines and rules defined in Chapter 6 were implemented. These guidelines and rules formed the model evaluation portion of OAKS. This model evaluation code provided the basis from

which to determine if the evolving problem model was consistent and complete. These evaluation functions were used by the model modification functions, which were the next portion of OAKS to be developed and implemented. These model modification functions made use of a pending issues list and an advisory issues list to keep the model consistent and complete, with respect to the guidelines and rules, throughout the problem model changes. Last, a windowed user interface was added to OAKS.

The OAKS system prototyped in this chapter has shown that a computer-based system that aids in the conduct of the OORA process is feasible and valuable. The OAKS system is based on an OORA mathematical model which embodies the basic principles of the OORA process. This model was implemented in OAKS in a form closely resembling the original mathematical model. All components and relationships in the model are embodied in the OAKS model. The OAKS system also captures the essence of a class by encapsulating the attributes and services in a class in a code structure that enforces that encapsulation. The domain-independent and domain-dependent guidelines and rules were implemented in such a way that they were usable as an evaluation tool on both the original OAKS domain model and the evolving problem model. The domain model can be checked using the LISP functions that implemented these guidelines and rules prior to the domain model's being modified. This ensures that the user starts with a consistent and complete model. It also provides a tool that can be used to check any OORA model before it is used in design and code. The evaluation functions are also used by the functions that implemented the model modification process to ensure that the changing model remained valid. Using a pending issues list allowed changes to be made to the model without adherence to any strict ordering of the changes, while ensuring that the model remained consistent and complete. The OAKS prototype has the features required to produce a sound model that represents an object-oriented specification of the system to be developed.

VIII. Conclusions and Recommendations

Conclusions

The software development method of OORA is one that is still maturing with many research questions still to be answered. This research has addressed an important gap in the development of processes and associated tools in the assistance of conducting an OORA. OAKS attacks the problem by developing a model that is truly object-oriented, and not a hybrid of processes (chapter 4), and evaluating that model based on concrete object-oriented criteria (chapters 5 and 6).

OAKS does not force formality in a process that is inherently informal. Informality will always exist during the requirements analysis process because this process is primarily cognitive, and deals with information that is uncertain and inconsistent. One way OAKS supports informality is by imposing minimum constraints on the order in which the components of the model must be acquired by the use of a list of issues that must be resolved before the model is considered complete. This list allows the user to make a majority of changes without regard to the order in which they are made, yet still maintain a valid model. Forcing the user to make changes in a rigid order could run counter to the developer's method of thinking (chapter 7).

Even though OAKS supports informality, the model developed is consistent and complete with respect to a defined set of OORA structures, relationships, guidelines and rules. The guidelines and rules consisted of those that were domain-independent based on the desired OORA components and relationships (chapter 6), those that were domain-dependent (chapter 6), and those that were based on the code structure of OAKS (chapter 7).

OAKS is not dependent on any particular domain for proper operation. It can be used in any domain where an OODA can be conducted. The results of an OODA are a set of object-oriented components and relationships. These components and relationships are modeled in the OAKS domain model which is based on the OORA mathematical model (chapter 4).

Specifically, the contributions of OAKS to the OORA process are the OORA math model, the OAKS domain model developed in CLOS, the guidelines and rules used to evaluate that model and the evolving problem model, the modification process used to create the problem model, and the modularity of OAKS, allowing changes or additions to the domain model or user interface without affecting the proper functioning of the system.

The OORA math model (chapter 4) provides a set of components and relationships that are critical to the development of an object-oriented system. These components are used in some form by a vast majority of the OORA processes and those that were required for an analysis of any object-oriented system. This math model can be used as a basis for the development of any OORA process; it was used as the basis for the OAKS domain and problem model.

The OAKS domain model, represented in CLOS (chapter 7), proved to be a very robust and flexible structure. The basic structure of the components in the generic domain model was easily used in developing the specific domain model for the particular domain chosen for the proof-of-concept in this research. The structure, though, is domain independent. Any domain in which an OODA can be conducted can be translated into the OAKS domain model. The generic domain model structure in OAKS contains all the components and relationships defined in the OORA math model. The classes in the generic domain model adhere to the object-oriented philosophies of a class and are encapsulated entities, with all structures and relationships in to the OORA math model contained in the classes. This is a significant change from the majority of other systems,

where other service information is treated differently than the remainder of the class components. The domain model also proved to be one that is easily analyzed for the form of the components and their adherence to any guidelines and rules. Because the domain structure is a separate file in the OAKS system, the domain information can be easily replaced with no effect on the operation of OAKS.

The guidelines and rules that were applied to the domain and the evolving problem model (chapters 6 and 7) proved to be extremely useful to both the problem model and the domain model. It is envisioned that the initial domain model is developed by conducting an OODA, the results of which have the form of Appendix A. This model is then translated into CLOS and inserted into OAKS. Once the domain model is in OAKS, the guidelines and rules would be extremely useful in identifying problems with the domain model prior to use by a user in OAKS. By applying the guidelines and rules to the domain model, violations are uncovered, thereby ensuring that the system starts with a consistent and complete domain model. Even if the domain model were never modified, OAKS could easily be used as a check to OORA results by entering those results into OAKS and testing them against the guidelines and rules. The process of entering the OORA results and the analysis of the guidelines and rules would provide an excellent check prior to going to OOD. The guidelines and rules were also essential to maintaining a consistent and complete model as the user modified the domain model to create a problem model.

The modification process in OAKS supports a relatively unstructured approach (chapter 7). Minimal restrictions were put on the order in which components of the model could be added, deleted or modified. This lack of restrictions on order would normally cause problems because the model would become inconsistent or incomplete if changes were not made in a certain order. However, this problem was overcome by using a list of pending issues. These pending issues are items that must be resolved before the model is considered complete. Each time a change is made that causes the model to become

inconsistent or incomplete in accordance with its guidelines and rules, an entry is added to pending issues. These entries are automatically removed when the problem was resolved. The user is always aware of these issues and can see what is added when certain changes are made. This form of system development is very supportive of the principles of good software engineering, yet it does not place the severe constraints of strict formality on the developer.

OAKS was coded using modules for the different code components so changes could be made without invalidating the system (chapter 7). The code modules were the domain model, the guidelines and rules, the modification procedures, and the user interface. The specific domain coded in the domain model or the user interface can be easily changed to a new domain or totally different user interface. Even changes in the guidelines and rules or modification procedures would not require any major changes to the system.

Recommendations

Even though OAKS addresses some of the current problems in OORA systems, there is much work left to be done. This work ranges from expanding the capabilities of OAKS to further evaluation of the potential of the OAKS system.

Dynamic properties of an OORA system were not addressed in OAKS. Even though a message trace through OAKS was possible, that was the extent of the analysis of any dynamic properties of the system. The only information gathered on state change within a class was data on which attributes may change as a result of the execution of a service. The elicitation of the dynamic properties of a class would require a special tool because of its nature. Research would have to be done on what form the information should take, how the information could be elicited, and how it could be integrated into OAKS.

A natural language parser would be desirable as an addition to OAKS. This would allow OAKS to more accurately identify possible relationships between classes in the system by looking at names that have the same meaning and knowing whether a name is a singular noun, or a verb, for example, according to the naming conventions of the model.

OAKS could incorporate learning, so that it learns from each problem model that is developed in a domain. Over time, the OAKS domain model would have to be modified to keep the model current. This modification would normally be done by an analyst in that domain. A research question is whether OAKS could track the changes that are made to the domain model to create the problem model and "learn" itself how the domain model is changing over time. This might allow OAKS to make changes to the domain model itself and keep itself current in the domain.

The user interface in OAKS, even though it is windows-based, is still not very sophisticated or robust. A better user interface could easily be attached to the OAKS system, replacing the current user interface. The current user interface simply uses LISP functions to modify the OAKS problem model. Some features missing from the existing user interface are the ability to select an item using the mouse without going through menu selections, a better way of presenting the list of pending issues, perhaps by highlighting the problem areas in the model, a better way of presenting the model itself using more graphical techniques, and better methods for getting user input

Related to the user interface is the development of a user's guide for OAKS. The information that would be placed in a user's guide on the changing of components and the creation of new components in the OAKS model has been already been defined in this research. This information should be supplemented with information on how to use the user interface.

Another area for further research is how to transform the problem model produced in OAKS into a formal specification or into a system for conducting OOD. This formal

specification could be used to go into the OOD phase or perhaps directly into code. The OAKS model seems to be conducive to this type of transformation.

OAKS should be used in testing in one domain for a variety of problems and also in a number of domain to evaluate its use across different domains.

Assistance could be built into OAKS for inserting the initial domain model. This would require the development of a user interface and process specifically for this task.

Even though there remains much to be done in this area of research, OAKS has created a solid foundation for future work. The basic structure of OAKS provides a sound yet flexible platform for expanding its capabilities with only minor changes to the existing OAKS structure. The concepts developed in OAKS and how those concepts are used provide an important contribution to the research in object-oriented requirements analysis.

Appendix A: Domain Model

The domain model chosen for this reserach was a system that manages the scheduling of maintenance and flight for aircraft squadron.

A squadron consists of flights of aircraft, personnel, facilities and a flight range. The personnel are the aircrew and the support personnel. The aircrew consist of pilots, navigators, and electronic warfare officers (EWO). Each support person is assigned to one shop. The facilities consist of maintenance hangars, spots on the flight line for parking aircraft, and the shops.

Aircraft maintenance is performed by the following shops:

1. A-shop (avionics)
2. B-shop (avionics)
3. C-shop (avionics)
4. Fuel
5. Hydraulics
6. Electrical and environmental
7. Egress
8. Propulsion
9. Machine shop
10. Corrosion
11. Non-destructive inspection (NDI)
12. Weapons
13. PMEL
14. ECM pods
15. AIS (maintains line replaceable units (LRU))
16. Parachute
17. Flight line personnel (includes crew chiefs)

18. Automatic ground equipment (AGE)

The scheduling of flights is done by the Plans and Scheduling shop.

There are two type of maintenance: maintenance required when a part is not operating correctly, and periodic inspection/maintenance.

Schedules must be kept on aircrew, aircraft flights, hangar use, flight line slot use, runway use, and range use.

The events that must be handled by the system are:

1. A part of a particular aircraft breaks.

   a. The repair is scheduled with the appropriate shop.

   b. The shop adds the broken part as a write-up for the aircraft, which may change the aircraft's status. The aircraft status can be fully mission capable, partly mission capable, or not mission capable.

   c. The shop determines the number of hours required for the repair and whether a maintenance hangar is needed.

   d. The shop schedules personnel to do the repair and a hangar, if necessary.

   e. If a hangar is necessary, the shop must wait until the hangar is available to start the repair.

2. A repair is complete.

   a. If the plane is in a hangar, it is moved to a spot on the flight line.

   b. The personnel that were assigned to the repair are released.

   c. The aircraft write-ups are updated and the aircraft status changed as necessary.

3. Schedule periodic maintenance.

   a. The periodic maintenance/inspection is scheduled with the appropriate shop.

   c. The shop determines the number of hours required for the repair and whether a maintenance hangar is needed.

d. The shop schedules personnel to do the work and a hangar, if necessary.

e. If a hangar is necessary, the shop must wait until the hangar is available to start the work.

4. The periodic maintenance/inspection is complete.

a. If the plane is in a hangar, it is moved to a spot on the flight line.

b. The personnel that were assigned to the work are released.

c. The aircraft maintenance log is updated.

5. Schedule a sortie.

a. The request for the sortie is given to the plans and scheduling shop. The request includes the desired date, aircraft required, their configurations, the aircrew required (by name or in general), the amount of time, and the part of the range required.

b. The sortie is scheduled and the information sent to the aircrew and the flight line shop.

6. Sortie complete

a. Update the schedule to show actual sortie information.

b. Update the number of hours on each aircraft part. This may require the scheduling of periodic maintenance.

c. Update the hours and types of missions for each aircrew.

7. Cancel a sortie.

The sortie is marked as cancelled.

The system must also handle the creation of a squadron. This will be done through the implied create service of every class. For this model, the squadron will be created and then used to create the remainder of the model.

This model is a very simplified one. It does not take into account such things as spare parts, repair done at the depot, etc. It also looks at components at a very high level. In an actual system, the aircraft components would be broken down into a smaller grouping of

components with each grouping containing information on how to repair, tools required, skills required, facilities required, repair times, and others. The model could easily be extended to the proper level.

The domain model is represented as a set of classes. The notation for the domain model is as follows:

list-of name ;; This is a list of all whose elements are of name

class ;; a class name

class-name.service ;; a service of that class

Class **squadron**

Superclass : none

Parts : **flight**, **aircraft-parking**, **aircrew**, **support-person**

Attributes:

name : string

flights : list-of **flight**

parking : **aircraft-parking**

aircrew : list-of **aircrew**

personnel : list-of **support-person**

Services:

change-name (new-name : name) return ()

pre : none

post :

(setf name new-name)

add-flight (new-flight : **flight**) return ()

pre : (not (member new-flight flights))

post :

(**flight**.create (new-flight))

(cons new-flight flights)

remove-flight (old-flight : **flight**) return ()

pre : (member old-flight flights)

post :

(**flight**.delete (old-flight))

(delete old-flight flights)

add-aircrew (new-person : **aircrew**) return ()

pre : (not (member new-person aircrew))

post :

(**aircrew**.create (new-person))

(cons new-person aircrew)

remove-aircrew (old-person : **aircrew**) return ()

pre : (member old-person aircrew)

post :

(**aircrew**.delete (old-person))

(delete old-person aircrew)

add-support (new-person : **support-person**) return ()

pre : (not (member new-person personnel))

post :

(**support-person**.create (new-person))

(cons new-person personnel)

remove-support (old-person : **support-person**) return ()

pre : (member old-person personnel)

post :

(**support-person**.delete (old-person))

(delete old-person personnel)

Class **flight**

Part-of : **Squadron**

Parts : **aircraft**

Attributes:

name : string

type-aircraft : string

the-aircraft : list-of **aircraft**

squadron : **squadron**

Services:

change-name (new-name : name) return ()

pre : none

post :

(setf name new-name)

change-type-aircraft (new-type : type-aircraft) return ()

pre : none

post :

(setf type-aircraft new-type)

add-aircraft (new-ac : **aircraft**)

pre : (not (member new-ac the-aircraft))

post :

(**aircraft**.create (new-ac))

(cons new-ac the-aircraft)

remove-aircraft (old-ac : **aircraft**)

pre : (member old-ac the-aircraft)

post :

(**aircraft**.delete (old-ac))

There is one instance of the class aircraft for every aircraft in the squadron

Class **aircraft**

Part-of : **flight**

Related to : **aircraft-schedule**

Parts : **aircraft-part**

Attributes :

model-number : string

tail-number : integer

the-flight : **flight**

status : (fully-mission-capable, partly-mission-capable, not-mission-capable)

inop-parts : list-of **aircraft-part**

;; inop-parts is the current list of parts of the aircraft that are inoperative.

schedule : **aircraft-schedule**

configuration : list-of **aircraft-part**

;; all the parts that are on the aircraft

Services :

in-op-part (ap : **aircraft-part**) return ()

pre : (member ap configuration)

post :

(cons ap inop-parts)

(possibly change status)

op-part (ap : aircraft-part) return ()

pre : (member ap inop-parts)

post :

(delete ap inop-parts)

(possibly change status)

get-tail-number () return integer

pre : none

post :

(tail-number)

get-flight () return flight

pre : none

post :

(the-flight)

get-status () return status

pre : none

post :

(status)

get-config () return configuration

pre : none

post :

(configuration)

add-part (new-part : **aircraft-part**) return ()

pre : (not (member new-part configuration))

post :

(**aircraft-part**.create (new-part))

(cons new-part configuration)

remove-part (old-part : **aircraft-part**) return ()

pre : (member old-part configuration)

post :

(**aircraft-part**.delete (old-part))

(delete old-part configuration)

get-sched () return aircraft-schedule

pre : none

post :

(schedule)

Class **aircraft-part**

Superclass : none

Part-of : **aircraft**

Related to : **support-shop**, **repair-symptoms**, **maintenance-history**, **periodic-maintenance**

Attributes:

part-name : string

a-aircraft : **aircraft**

;; There is an object created for each part on each aircraft. Each object is uniquely identifed by the aircraft it is part of.

number-of-flight-hours : integer

;; The number of hours on the part

repair-shop : **support-shop**

current-symptoms : **repair-symptoms**.legal-symptoms-list

status : (operative, need-repair)

symtom-analysis : **repair-symptoms**

history : **maintenance-history**

periodic : **periodic-maintenance**

Services:

get-aircraft () return **aircraft**

pre : none

post :

(a-aircraft)

get-part-name () return string

pre : none

post :

(part-name)

get-sym-analysis () return repair-symptoms

pre : none

post :

(symptom-analysis)

inoperative (symptoms : **repair-symptoms**.legal-symptoms-list) return ()

pre : none

post :

;; If the current status is operative, then the repair of the part must be scheduled. If not, the part is already scheduled for repair so add the new symptoms to the list. When the part is being repaired, the technician will look at the current symptoms list.

(if (eql status 'operative))

(repair-shop.schedule-repair (**aircraft-part**))

(a-aircraft.in-op-part (**aircraft-part**)

(setf status 'need-repair))

(cons symptoms current-symptoms)

repaired (type : maintenance-history.type) return ()

pre : none

post :

(if (eql type 'fix)

(setf status 'operative)

(a-aircraft.op-part (aircraft-part)

(setf current-symptoms '())))

(history.add-maintenance (date, type))

update-flight-hours (new-hours : integer) return ()

pre : none

post :

(setf number-of-flight-hours (+ number-of-flight-hours new-hours))

(periodic.check-list (number-of-flight-hours))

current-symptoms-list () return **repair-symptoms**.legal-symptom-list

pre : none

post :

(current-symptoms)

shop-name () return **support-shop**

pre : none

post :

(repair-shop)

Class **periodic-task**

Superclass : none

Part-of : **periodic-maintenance**

Attributes:

part-name : **aircraft-part**

hours : integer

;; number of hours on a part before the task is to be done.

hangar-required : boolean

task-name : string

Services:

hours-to-task () return hours

pre : none

post :

(hours)

change-time-before-repair (new-hours : integer) return ()

pre : none

post :

((setf hours new-hours

hangar-needed () return boolean

pre : none

post :

(hangar-required)

Class **periodic-maintenance**

Superclass : none

Related to: **aircraft-part**

Parts : **periodic-task**

Attributes:

part : **aircraft-part**

task-list : list-of **periodic-task**

Services:

check-list (new-hours : integer) return ()

;; when a part has its number of hours updated, the periodic maintenance list is checked. If maintenance needed, it is scheduled.

pre : none

post :

(let ((result '())

(dolist (task task-list)

(if (< task.hours-to-task new-hours)

(cons task result)))

(if result

(let ((the-shop part.shop-name))

(dolist (one-task result)

(the-shop.schedule-periodic (part-name, one-task)

;; There is one object for each type of part, not for each aircraft.

Class **repair-symptoms**

Superclass : none

Related to : **aircraft-part**

Attributes:

part-name : **aircraft-part**

legal-symptoms-list : enumerated-list

Services:

determine-hangar-need (symptoms : legal-symptoms-list) return boolean

pre : none

post :

(return true or false based on the current symptoms)

;; There is one for each part on each aircraft

Class **maintenance-history**

Superclass : none

Related to : **aircraft-part**

Attributes:

part : **aircraft-part**

type : **support-shop**.type

history-list : list-of (date : **schedule-event**.day, type : **support-shop**.type)

Services:

add-maintenance (date : schedule-event.day, a-type : support-shop.type) return ()

pre : none

post :

(cons '(date a-type) history-list)

Class **people**

Superclass : none

Attributes :

name : string

ssan : integer

squad : **squadron**

AFSC : string

Services :

change-squadron (new-squadron : ***squadron***) *return* ()

pre : none

post :

(setf squad new-squadron)

Class **support-person**

Superclass : people

Part-of : **squadron**

Attributes:

shop : **support-shop**

type : **support-shop**.type

jobs-to-do : list-of (**aircraft-part**, **support-shop**.type)

Services:

job-list () return list-of-jobs

pre : none

post :

(jobs-to-do)

add-job (new-job : aircraft-part, the-type : type) return ()

pre : none

post :

(cons '(aircraft-part type) jobs-to-do)

remove-job (job : aircraft-part; the-type : type) return ()

pre : none

post :

(delete '(job the-type) jobs-to-do)

Class **support-shop**

Superclass : none

Related to : aircraft-part

Attributes :

list-of-people : list-of **support-person**

shop-name : (a-shop, b-shop, c-shop, fuel-shop, hydraulics, electrical-and-environmental, egress, propulsion, machine-shop, corrosion-control, non-destructive-inspection, weapons, PMEL, LRU, parachute, flight-line-support, aircraft-ground-equipment)

type : (fix, **periodic-task**)

jobs-pending : list-of (**aircraft-part**, type)

jobs-in-hangars : list-of (**aircraft-part**, **hangar**, type)

Services:

schedule-repair (ap : **aircraft-part**) return ()

pre : none

```
    post:
        (let ((sym ap.get-sym-analysis))
          (if (sym.determine-hangar-need (ap.current-symptoms-list))
            ;; then
            ((aircraft-parking.schedule-hangar (support-shop, ap, fix))
             (append jobs-pending '(ap fix)))
             ;; else
             (let ((the-person (first list-of-people))
                 (dolist (a-person list-of-people)
                   (if (< (length a-person.job-list) (length the-person.job-list))
                     (setf the-person a-person)))
                 (the-person.add-job(ap fix)))))
hangar-available (ha.  : hangar; ap : aircraft-part; a-type : type) return ()
    pre : (not (eql jobs-pending '()))
    post :
        (let ((the-person (first list-of-people))
            (dolist (a-person list-of-people)
              (if (< (length a-person.job-list) (length the-person.job-list))
                (setf the-person a-person)))
          (the-person.add-job(ap a-type)))
        (delete ap jobs-pending)
        (cons (ap han) jobs-in-hangar)
schedule-periodic (ap : aircraft-part; pt : periodic-task) return ()
    pre : none
    post :
        (if (pt.hangar-needed)
```

```
                ;; then
                ((aircraft-parking.schedule-hangar (support-shop, ap, pt))
                 (append jobs-pending '(ap pt)))
                     ;; else
                 (let ((the-person (first list-of-people))
                     (dolist (a-person list-of-people)
                         (if (< (length a-person.job-list) (length the-person.job-list))
                             (setf the-person a-person)))
                     (the-person.add-job(ap pt)))))
repair-complete (ap : aircraft-part; the-type : type) return ()
    pre : none
    post :
        ;; free up the person
        (dolist (a-person list-of-people)
          (dolist (a-job a-person.job-list)
              (if (and (member ap a-job)
                       (member the-type a-job))
                  (a-person.remove-job (ap the-type))))
        ;; change status of part
        (ap.repaired (the-type))
        ;; release hangar, if in hangar, and get a free flight line slot
        (let ((hangar '()))
          (if (eql (first job-in-hangars) ap)
              (setf hangar (second job))
          (if hangar
              (aircraft-parking.release-hangar (hangar))
```

(delete (ap the-type) jobs-in-hangars)

Class **aircrew**

Superclass : people

Part-of : **squadron**

Related to : **aircrew-schedule**

Attributes:

type : (pilot, navigator, EWO)

aircraft-checked-out-in : **aircraft**

hours : integer

schedule : **aircrew-schedule**

Services :

get-sched () return **aircrew-schedule**

pre : none

post:

(schedule)

update-hours (new-hours : hours) return ()

pre : none

post :

(setf hours (+ hours new-hours))

Class **mission**

Related to : **plans-and-scheduling**

Attributes:

date : integer

mission-type : (test, eval)

ac-info : list-of (aircraft, schedule-event.duration, aircraft.configuration)

;; these hours are those needed for each aircraft

aircrew-list : list-of (aircrew, aircraft)

time : (schedule-event.start-time, schedule-event.duration)

;; this is the time for the entire mission

range-info : (real, int)

status : (cancelled, complete)

Services:

get-aircraft () return list-of **aircraft**

pre : none

post :

```
(let ((ac-list '()))
    (dolist (an-ac ac-info)
        (cons (first an-ac) ac-list))
    (ac-list))
```

get-duration () return time

pre : none

post :

```
(second time)
```

get-date () return date

pre : none

post:

```
(date)
```

get-config (ac : aircraft) return aircraft.configuration

pre : none

post :

```
(dolist (an-ac ac-info)
    (if (eql (first an-ac ac))
```

```
            (return (third an-ac)))
all-aircrew () return aircrew-list
    pre : none
    post :
        (aircrew-list)
get-mission-type () return mission-type
    pre : none
    post :
        (mission-type)
get-range-info () return range-info
    pre : none
    post :
        (range-info)
get-aircrew () return list-of-aircrew
    pre : none
    post :
        (let ((ac-list '()))
            (dolist (one-crew aircrew-list)
                (cons (first one-crew) ac-list))
            (ac-list))
change-date (new-date : date) return ()
    pre : none
    post :
        (setf date new-date)
change-time (new-time : time) return ()
    pre : none
```

post :

(setf time new-time)

change-ac-info (new-ac-info : ac-info) return ()

pre : none

post :

(setf ac-info new-ac-info)

change-aircrew-list (new-aircrew-list : aircrew-list) return ()

pre : none

post :

(setf aircrew-list new-aircrew-list)

change-status (new-status : status) return ()

pre : none

post :

(setf status new-status)

Class **plans-and-scheduling**

Related to : **mission**

Attributes :

range : **range-schedule**

missions : list-of **mission**

Services :

mission-request (ac-list : (aircraft, aircraft.configuration), list-of-aircrew : list-of aircrew, schedule-event.duration, mission.range-info) return ()

pre : none

post :

;; schedule the mission based on all the existing schedules.

(**aircrew**.get-sched)

(**aircraft**.get-sched)

(range)

;; create the mission

(cons (**mission**.create (date, type, time, ac, aircrew, time, range) missions)

;; schedule aircraft

(dolist (ac-info ac)

((first ac-info).get-sched.add-mission (date, (first time), (second time), (third ac-info)))

;; schedule aircrew

(dolist (arc aircrew)

((first arc).get-schedule.add-mission (date, (first time), (second time), (second arc), type)))

;; schedule range

(let ((acl '()))

(dolist (an-ac ac)

(cons (first an-ac) acl))

(range.add-mission (date, time, acl, range)))

mission-complete (the-mission : **mission,** hours : mission.ac-info, crew : mission.aircrew-list, date : integer, time : mission.time) return ()

pre : (member mission missions)

post :

;; update hours on aircraft parts and aircrew hours

(dolist (each-ac hours)

(dolist (each-part (first each-ac).get-config)

(each-part.update-flight-hours (second.each-ac)))

(dolist (a-crew crew)

(if (member (first each-ac) a-crew)

(a-crew.update-hours (second each-ac)))))

;; update mission info

(the-mission.change-date (date))

(the-mission.change-time (time))

(the-mission.change-ac-info (hours))

(the-mission.change-aircrew-list (crew))

(the-mission.change-status (complete))

cancel-mission (the-mission : mission) return ()

pre : (member the-mission missions)

post :

(the-mission.change-status (cancelled)

;; cancel for an aircraft

(dolist (ac the-mission.get-aircraft)

((ac.get-sched).remove-mission (mission.get-date (first mission.get-duration), (second mission.get-duration), mission.get-config(ac))))

;; cancel for aircrew

(dolist (an-aircrew mission.all-aircrew)

((first (an-aircrew.get-sched)).remove-mission (mission.get-date, (first mission.get-duration), (second mission.get-duration), (second an-aircrew), mission.get-mission-type)))

;; cancel range

(range.remove-mission (mission.get-date, mission.get-duration, mission.get-aircraft, mission.get-range-info)

Class **schedule-event**

attributes:

day : integer

start-time : real

duration : real

services :

get-day () return integer

pre : none

post :

(day)

get-start () return real

pre : none

post :

(start-time)

get-duration () return real

pre : none

post :

(duration)

Class **aircrew-schedule-event**

Superclass : **schedule-event**

Part of : **aircrew-schedule**

Attributes :

type-of-mission : (test, eval)

the-aircraft : **aircraft**

Services :

get-type () return (test, eval)

pre : none

post :

(type-of-mission)

get-aircraft () return **aircraft**

pre : none

post :

(the-aircraft)

Class **aircraft-schedule-event**

Superclass : **schedule-event**

Part of : **aircraft-schedule**

Attributes :

configuration : aircraft.configuration

Services :

get-config () return config

pre : none

post :

(configuration)

Class **range-schedule-event**

Superclass : **schedule-event**

Part of : **plans-and scheduling**.range

Attributes :

ac : list-of **aircraft**

range-use : mission.range-info

Services :

get-aircraft () return list-of-**aircraft**

pre : none

post :

(ac)

get-range-info () return list-of (altitudes, airspace, facilities)

pre : none

post :

(range-use)

Class **aircrew-schedule**

Related to : **aircrew**

Parts : **aircrew-schedule-event**

Attributes :

schedule : list-of **aircrew-schedule-event**

the-aircrew : **aircrew**

Services:

add-mission (day : **aircrew-schedule-event**.day, start-time : **aircrew-schedule-event**.start-time, duration : **aircrew-schedule-event**.duration, an-aircraft : **aircraft**, mission-type : **aircrew-schedule-event**.type-of-mission) return ()

pre : none

post :

(cons (**aircrew-schedule-event**.create(day, start-time, duration, an-aircraft, mission-type)) schedule)

remove-mission (day : **aircrew-schedule-event**.day, start-time : **aircrew-schedule-event**.start-time, duration : **aircrew-schedule-event**.duration, an-aircraft : **aircraft**, mission-type : **aircrew-schedule-event**.type-of-mission) return ()

pre : none

post :

(dolist (one-event schedule)

(if (and (eql one-event.get-day day)

(eql one-event.get-start start-time)

(eql one-event.get-duration duration)

(eql one-event.get-aircraft an-aircraft)

(eql one-event.get-type mission-type))

((delete one-event schedule)

(**aircrew-schedule-event**.delete (one-event))))

get-sched () return list-of **aircrew-schedule-event**

pre : none

post :

(schedule)

Class **aircraft-schedule**

Related to : **aircraft**

Part : **aircraft-schedule-event**

Attributes :

schedule : list-of **aircraft-schedule-event**

the-aircraft : **aircraft**

Services:

add-mission (day : **aircraft-schedule-event**.day, start-time : **aircraft-schedule-event**.start-time, duration : **aircraft-schedule-event**.duration, config : **aircraft**.configuration) return ()

pre : none

post :

(cons (**aircraft-schedule-event**.create(day, start-time, duration, config)) schedule)

remove-mission (day : **aircraft-schedule-event**.day, start-time : **aircraft-schedule-event**.start-time, duration : **aircraft-schedule-event**.duration, config : **aircraft-schedule-event**.configuration) return ()

pre : none

post :

(dolist (one-event schedule)

(if (and (eql one-event.get-day day)

(eql one-event.get-start start-time)

(eql one-event.get-duration duration)

(eql one-event.get-config config))

((delete one-event schedule)

(**aircraft-schedule-event**.delete (one-event))))

get-sched () return list-of **aircraft-schedule-event**

pre : none

post :

(schedule)

Class ***range-schedule***

Parts : **range-schedule-event**

Attributes :

schedule : list-of **range-schedule-event**

Services :

add-mission (day : **range-schedule-event**.day, time : **mission**.time, aircraft : **range-schedule-event**.ac, range-info : **range-schedule-event**.range-use) return ()

pre : none

post :

(cons (**range-schedule-event**.create (day, (first time), (second time), aircraft, range-info)) schedule)

remove-mission (day : **range-schedule-event**.day, time : **mission**.time, aircraft : **range-schedule-event**.ac, range-info : **range-schedule-event**.range-use) return ()

pre : none

post :

(dolist (one-event schedule)

(if (and (eql one-event.get-day day)

(eql (one-event.get-start one-event.get-duration) time)

(eql one-event.get-aircraft aircraft)

(eql one-event.get-range-info range-info))

((delete one-event schedule)

(**range-schedule-event**.delete (one-event))))

get-sched () return list-of **range-schedule-event**

pre : none

post :

(schedule)

Class **aircraft-parking**

Superclass : none

Part of : **squadron**

Parts : **hangar**, **flight-line-spots**

Attributes :

spots : list-of **flight-line-spots**

hangars : list-of **hangar**

type : **maintenance-history**.type

requests-pending : list-of (**support-shop**, **aircraft-part**, **maintenance-history**.type)

Services:

add-hangar (new-hangar : **hangar**) return ()

pre : (not (member new-hangar hangars))

post :

(**hangar**.create (new-hangar))

(cons new-hangar hangars)

remove-hangar (old-hangar : **hanger**) return ()

pre : (member old-hangar hangars)

post :

(**hangar**.delete (old-hangar))

(delete old-hangar hangars)

release-hangar(the-hangar : **hangar**) return (**flight-line-spots**)

pre : none

post :

(the-hangar.release (the-aircraft))

(if requests-pending

(let ((fill (first requests-pending)))

((first fill).hangar-available (the-hangar, (second fill), (third fill))

(the-hangar.new-aircraft ((second fill).get-aircraft))

(delete fill requests-pending)

(dolist (a-spot spots)

(if (not (a-spot.occupied))

(a-spot.fill (the-aircraft))

(return a-spot)))))

schedule-hangar (ss : **support-shop**, ap : **aircraft-part,** type : **maintenance-history**.type) return ()

pre : none

post:

(let ((avail '()))

(dolist (h hangars)

```
                (if h.available
                    (cons h avail)))
            (if avail
            ;; then
              (((first h).new-aircraft (ap.get-aircraft))
              (ss.hangar-available ((first h) ap type))
               ;; release flight line spot
               (let ((ac ap.get-aircraft))
                    (dolist (a-spot spots)
                        (if (eql a-spot.occupied ac)
                            (a-spot.empty)
                            (return))))
            ;;else
              (append  requests-pending '((ss ap type)))))
```

add-spot (new-spot : **flight-line-spots**) return ()

pre : (not (member new-spot spots))

post :

(cons (**flight-line-spots**.create (new-spot)) spots)

delete-spot (old-spot : **flight-line-spots**) return ()

pre : (member old-spot spots)

post :

(delete old-spot spots)

(**flight-line-spots**.delete (old-spot))

Class **hangar**

Superclass : none

Part of : **aircraft-parking**

Attributes:

occupied-by : **aircraft**

Services:

new-aircraft (ac : **aircraft**) return ()

pre : none

post :

(setf occupied-by aircraft)

available () return boolean

pre : none

post :

(if occupied-by

'()

t)

release () return **aircraft**

pre : none

post :

(let ((ac-in occupied-by))

(setf occupied-by '())

(ac-in))

Class **flight-line-spots**

Superclass : none

Part-of : **aircraft-parking**

Attribute:

the-aircraft : **aircraft**

Services :

occupied () return **aircraft**

pre : none

post :

(the-aircraft)

fill (ac : **aircraft**) return ()

pre : (null the-aircraft)

post :

(setf the-aircraft ac)

empty () return **aircraft**

pre : none

post :

(let ((ac the-aircraft))

(setf the-aircraft '())

(ac))

The following is a tracing of each of the major events discussed in the beginning of the appendix. These traces show the use of the various classes and services in the model. The format is "class-name.service-name". The leftmost class and service name calls the class and service names indented beneath it.

Event A - Part Breaks

aircraft-part.inoperative (symptoms)

support-shop.schedule-repair (aircraft-part)

aircraft.in-op-part (aircraft-part)

support-shop.schedule-repair (aircraft-part)

aircraft-part.get-sym-analysis

repair-symptoms.determine-hangar-need (symptoms)

aircraft-part.current-symptoms-list

aircraft-parking.schedule-hangar (support-shop, aircraft-part, fix)

support-person.job-list

support-person.add-job (aircraft-part, fix)

aircraft-parking.schedule-hangar (support-shop, aircraft-part, fix)

hangar.available

hangar.new-aircraft (aircraft)

support-shop.hangar-available (hangar, aircraft-part, type)

aircraft-part.get-aircraft

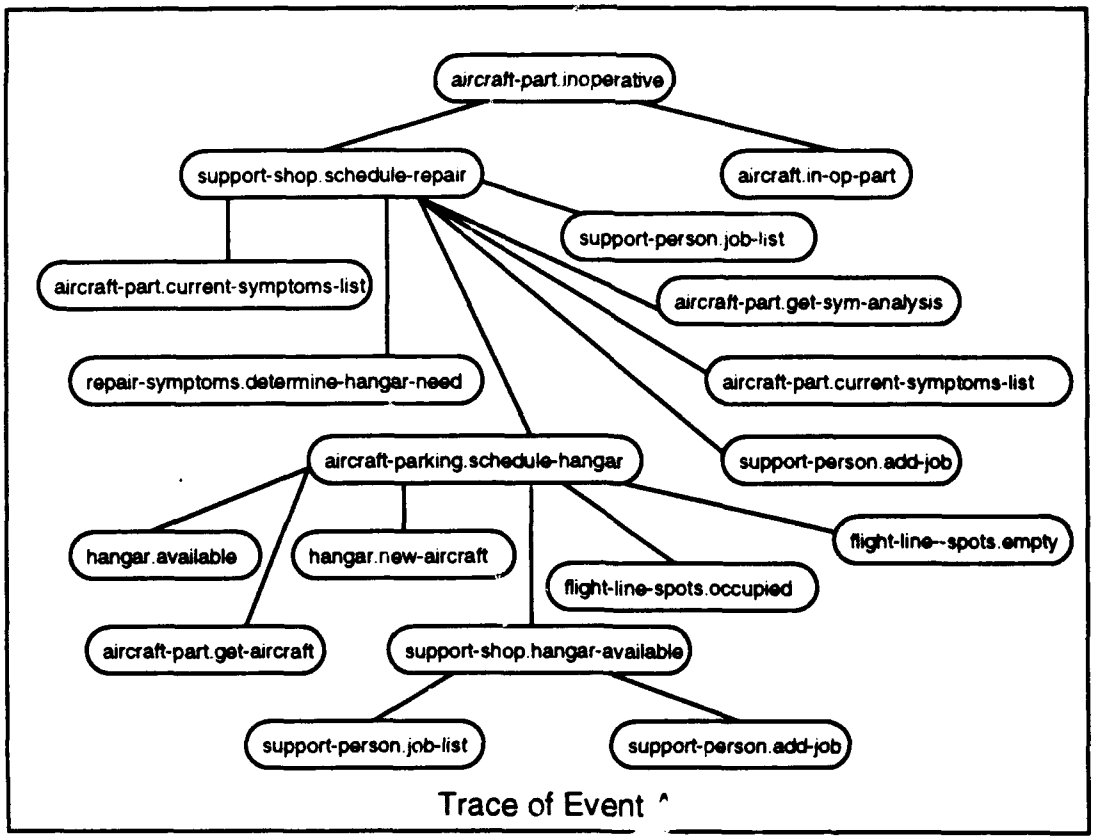
flight-line-spots.occupied

flight-line-spots.empty

support-shop.hangar-available (hangar, aircraft-part, type)

support-person.job-list

support-person.add-job



Event B - Part Fixed

support-shop.repair-complete (aircraft-part, type)

support-person.job-list

support-person.remove-job (aircraft-part, type)

aircraft-part.repaired

aircraft-parking.release-hangar (hangar)

aircraft-part.repaired

aircraft.op-part(aircraft-part)

maintenance-history.add-maintenance (date, type, aircraft-part)

aircraft-parking.release-hangar (hangar)

hangar.release (aircraft)

support-shop.hangar-available (hangar, aircraft-part, type)

flight-line-spots.occupied

flight-line-spots.fill

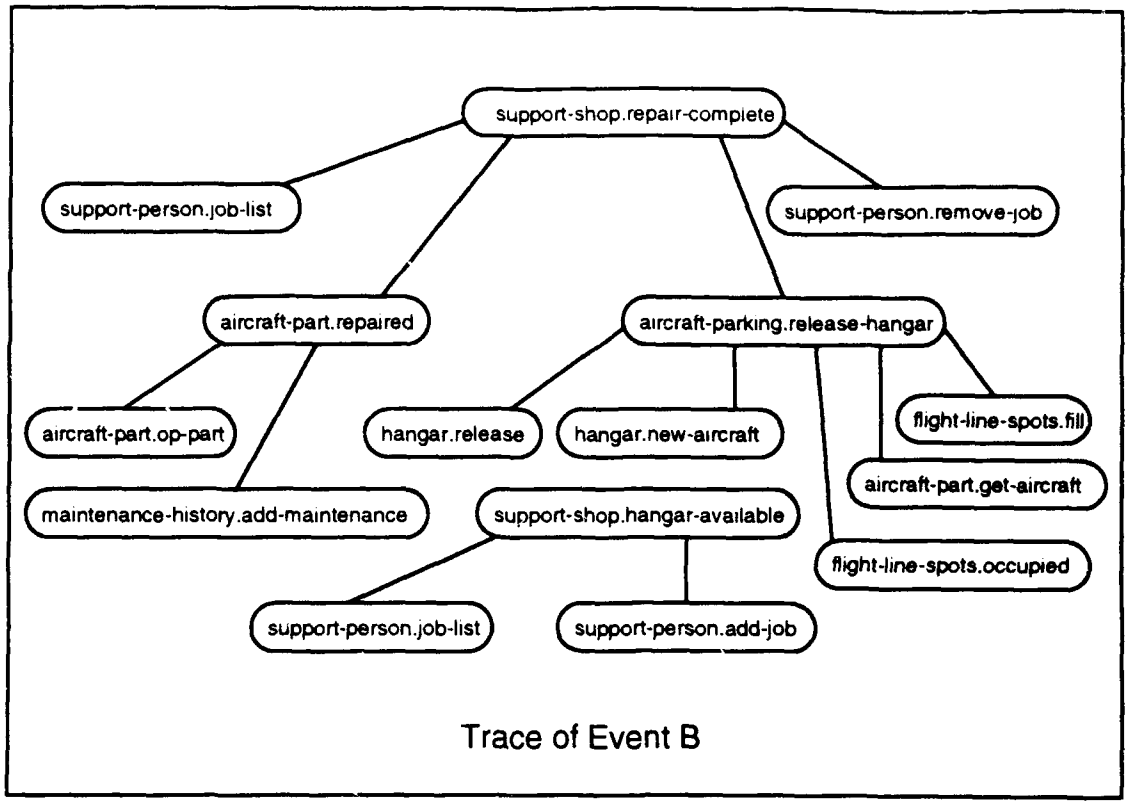
hangar.new-aircraft (aircraft)

aircraft-part.get-aircraft

support-shop.hangar-available (hangar, aircraft-part, type)

support-person.job-list

support-person.add-job



Trace of Event B

Event C - Schedule Mission

plans-and-scheduling.mission-request (ac-list, aircrew-list, time-range, range-info)

aircrew.get-sched

aircraft.get-sched

mission.create

aircraft-schedule.add-mission (date, start, duration, config)

aircrew-schedule.add-mission (date, start, duration, aircraft, type)

range-schedule.add-mission (date, time, aircraft-list, range)

aircraft-schedule.add-mission (date, start, duration, config)

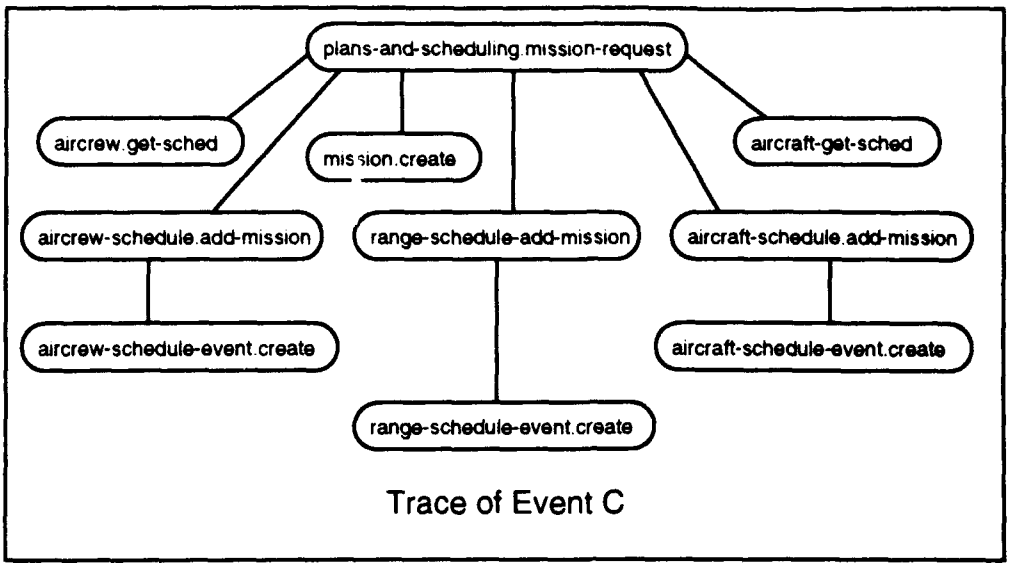
aircraft-schedule-event.create (day, start-time, duration, config)

aircrew-schedule.add-mission (date, start, duration, aircraft, type)

aircrew-schedule-event.create (day, start-time, duration, aircraft, mission-type)

range-schedule.add-mission (date, time, aircraft-list, range)

range-schedule-event.create (day, (first time), (second time), ac, range-info)



Trace of Event C

Event D - Mission Complete

plans-and-scheduling.mission-complete (mission, hours, crew, date, time)

aircraft.get-config

aircraft-part.update-flight-hours (hours)

aircrew.update-hours (hours)

mission.change-date (date)

mission.change-time (time)

mission.change-ac-info (ac-info)

mission.change-aircrew-list (aircrew-list)

mission.change-status (status)

aircraft-part.update-flight-hours (hours)

periodic-maintenance.check-list (hours)

periodic-maintenance.check-list (hours)

periodic-task.hours-to-task

aircraft-part.shop-name

support-shop.schedule-periodic (part, one-task)

support-shop.schedule-periodic (part, one-task)

periodic-task.hangar-needed

aircraft-parking.schedule-hangar (support-shop, part, periodic-task)

support-person.job-list

support-person.add-job (part, periodic-task)

aircraft-parking.schedule-hangar (support-shop, aircraft-part, periodic-task)

hangar.available

hangar.new-aircraft (aircraft)

support-shop.hangar-available (hangar, aircraft-part, type)

aircraft-part.get-aircraft

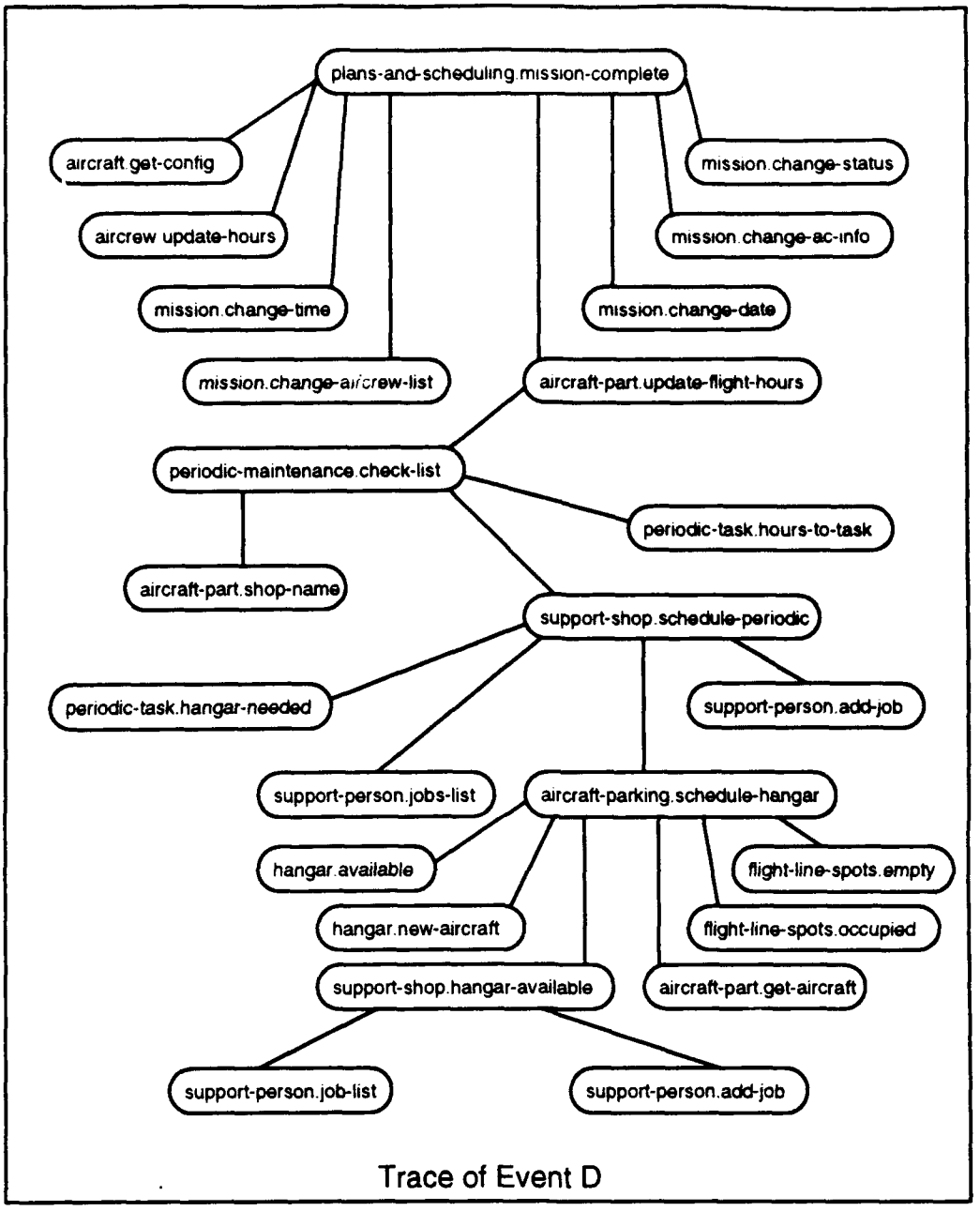
flight-line-spots.occupied

flight-line-spots.empty

support-shop.hangar-available (hangar, aircraft-part, type)

support-person.job-list

support-person.add-job



Trace of Event D

Event E - Mission Is Cancelled

plans-and-scheduling.cancel-mission

aircraft-schedule.remove-mission

mission.get-date

mission.get-duration

mission.get-config (ac)

aircrew-schedule.remove-mission

mission.get-mission-type

range-schedule-remove-mission

mission.get-aircraft

mission.get-range-info

mission.change-status

aircraft.get-sched

mission.all-aircrew

aircrew.get-sched

aircraft-schedule.remove-mission

aircraft-schedule-event.get-day

aircraft-schedule-event.get-start

aircraft-schedule-event.get-duration

aircraft-schedule-event.get-config

aircraft-schedule-event.delete (one-event)

aircrew-schedule.remove-mission

aircrew-schedule-event.get-day

aircrew-schedule-event.get-start

aircrew-schedule-event.get-duration

aircrew-schedule-event.get-aircraft

aircrew-schedule-event.get-type

aircrew-schedule-event.delete (one-event)

range-schedule.remove-mission

range-schedule-event.get-day

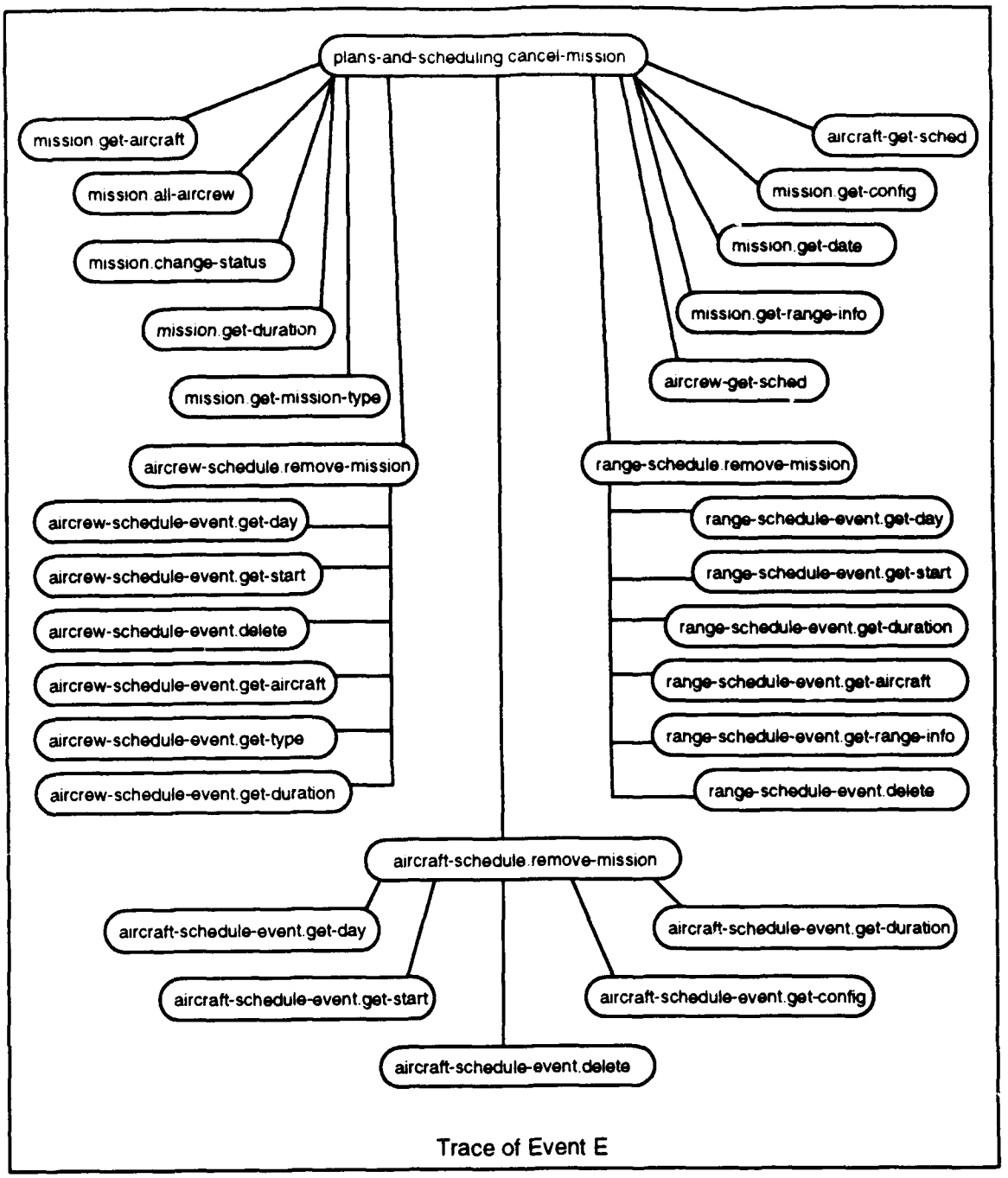
range-schedule-event.get-start

range-schedule-event.get-duration

range-schedule-event.get-aircraft

range-schedule-event.get-range-info

range-schedule-event.delete (one-event)



Trace of Event E

Appendix B: Example OAKS Session

This appendix contains the steps involved in an OAKS session where the user adds a new class into the model. The class that is added is the class "maintenance-equipment", which is the equipment used to maintain the aircraft in the squadron. The class "maintenance-equipment" is a part of the class "squadron" and will contain the attribute of "name" and the service "change-name".

In the appendix, the following conventions are used:

The selection of a menu item with the right mouse button is shown in italics, as in *Menu-Item*.

Text output by the OAKS system is shown as underlined text, as in OAKS Text.

Text typed in by the user is shown in bold, as in **User Input**.

The session is started by initiating a SUN Common LISP environment containing the LISPView and CLOS packages. At the LISP prompt, the user would type **(load "oaks.lisp")**. This file loads the files "oaksd.lisp", which contains the domain model structure and the domain model, "oaksno.lisp" which contains the model evaluation procedures, "oaksmod.lisp" which contains the model modification procedures, "oaksave.lisp" which saves the changed model to a file, executes the procedure "read-data", which reads from the file "userfil", and then loads "oaksui.lisp" which contains the LISPView user interface. The file "userfil" contains the problem model. This is the model the user modifies to create a model for the particular problem of interest. When OAKS is first used, "userfil" contains the unmodified domain model.

The user then types **(in-package 'oaks)** at the LISP prompt. All the files are loaded into this package.

The user interface creates the LISPView OAKS window as described in Chapter 7 and shown in Figure 7-2. The three windows as shown in that figure will be referenced as well as the menus.

When the LISPView OAKS window is first created, window 1 contains a list of all classes in the model including the parents of the classes, window 2 is blank, and window 3 contains the pending issues. If this is the first time the user has used OAKS, the only entry in pending issues will be (CLASSES NEED VERIFIED), which will remain in the pending issues list until all classes in the model have been verified.

The user now will operate exclusively in the LISPView OAKS window.

The user wants to add a new class to the model called "maintenance-equipment" which is a part of the class "squadron". The user can either first create the new class, or first put the new whole/part structure in the "squadron" class and then create the new class. The order of the operations will not make any difference to the final outcome. The user decides to first place the new whole/part structure in the "squadron" class.

Model/Class

squadron

The user uses the Model/Class menu to select the "squadron" class. Window 1 now contains the information on the class "squadron", which includes the class name, description, the attribute names, the service names, the whole/part structure which includes class names and ranges, the relation structure which includes the relation name, class names and ranges, the parents of the class, and whether or not the class is verified. Windows 2 and 3 are unchanged.

Component

Whole-Part

The component menu is used to select the whole-part component of the "squadron" class. Windows 1 and 3 are unchanged, but window 2 now contains the whole/part structures of the "squadron" class.

Action

Add Whole/Part Relation

The action menu is used to add a new whole/part relation to the "squadron" class. At this point, a pop-up box appears to gather the information required from the user. The user goes to the first piece of requested information by clicking on the line after "Class1" using the left mouse button. The user can move between fields in the pop-up box by using the up and down arrow keys. The following are the fields requested and the user input.

Class1 **squadron**

Range1 **(1 n)**

Class2 **maintenance-equipment**

Range2 **(1 1)**

The user then presses the "Done" button using the left mouse button and the pop-up box disappears and the changes are made to the model.

Windows 1 and 2 now show the new whole/part relation. Window 3 contains a new pending issues entry:

(MISSING-CLASS-AND-RELATION MAINTENANCE-EQUIPMENT #S(RELATION NAME WHOLE/PART CLASS1 SQUADRON RANGE1 (1 N) CLASS2 MAINTENANCE-EQUIPMENT RANGE2 (1 1))

This indicates that one of the classes in the new relation, "maintenance-equipment", does not currently exist in the model.

The next step is to add the new class.

Model/Class

Entire Model

The entire model must be selected in order to add a new class. This changes window 1 back to showing the classes in the model, window 2 is blank and window 3 is unchanged.

Action

Add a Class

The pop-up box is filled out as follows.

Class name **maintenance-equipment**

Class description **Equipment required to repair aircraft.**

The "Done" button is pressed. Window 1 is changed to include the "maintenance-equipment" class. Window 2 is still blank. In window 3, the pending issue on the missing class and relation is removed. The OAKS system automatically added the whole/part relation to the new class. This can be seen by the user doing the following actions.

Model/Class

maintenance-equipment

Window 1 now contains the "maintenance-equipment" class, and the whole/part slot will contain the relation with the "squadron" class.

The next step is to add the "name" attribute.

Action

Add an Attribute

The pop-up box is filled out as follows.

Name **name**

Desc **The name of the equipment**

Base value **str**

Lower value (opt)

Upper value (opt)

The information for lower and upper value is left blank and "Done" is selected.

Window 1 changes to show the new attribute name. To see the entire attribute structure, the following is done.

Component

One Attribute

NAME

Windows 1 and 3 are unchanged, but window 2 now contains the attribute structure containing the name, description, initial value, legal values to include the base, lower and upper values, and whether or not the attribute is verified. Since this attribute was created by the user, it is automatically set as verified.

The last step is to add the service that changes the "name" attribute.

Component

Entire Class

Windows 1 and 3 are the same, but window 2 is now blank.

Action

Add Service Using Template

Since the service changes the value of an attribute, one of the four service templates can be used. These templates save the user time and effort by filling out many of the slots in the service automatically.

The pop-up box is filled out as follows:

Template (change, return, add, remove) **change**

Attribute name **name**

Service name **change-name**

After "Done" is selected, the new service is created. Window 1 will now contain the service name. To see the new service:

Component

One Service

CHANGE-NAME

Windows 1 and 3 are unchanged. Window 2 contains the service and its components. The components of the service are its name, description, input parameters, output parameters, precondition, postcondition changed attributes, postcondition messages and whether or not the service is verified. The service template automatically filled in all the information not supplied by the user, which was the description, input parameters, output parameter, precondition and the postcondition. The service is shown as verified because it was created by the user.

To save the changes made thus far, the user would select the "Save" menu item.

Appendix C: OAKS Code

Due to its size, Appendix C was not attached to this report. The appendix is distributed separately as an Air Force Institute of Technology Technical Report, AFIT/EN/TR/93-07.

Bibliography

[ARAN89] Arango, Guillermo, "Domain Analysis - From Art Form to Engineering Discipline," *SIGSOFT Engineering Notes*, Vol 14(3), May 89, p. 152-159.

[ARIN89] Arinze, Bay, "A Natural Language Front End for Knowledge Acquisition," *SIGART Newsletter*, No. 108, Apr 89, p. 106-114.

[BABB85] Babb, Robert B.; Kieburtz, Richard; et al, "Workshop on Models and Languages for Software Specification and Design," *Computer*, Vol 18, No 3, Mar 85, p. 103-108.

[BAIL89] Bailin, Sidney C., "An Object-Oriented Requirements Specification Method," Communications of the ACM, Vol 32, No 5, May 89, p. 608-623.

[BALZ78] Balzer, Robert; Goldman, Neil; Wile, David, "Informality in Program Specifications," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2, Mar 78, p. 94-103.

[BALZ79] Balzer, Robert; Goldman, Neil, "Principles of Good Software Specification and Their Implications for Specification Language," *Proceedings of the Specifications for Reliable Software Conference*, Apr 79, p. 58-67.

[BALZ85] Balzer, Robert, "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, p. 1257-1267.

[BARS85] Barstow, David R., "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, Nov 85, p. 1321-1336.

[BERA92a] Berard, Ed, "Object-Oriented Requirements Analysis," Unpublished, Received through E-mail Jan 92, Berard Software Engineering, Inc, (301)417-9884.

[BERA92b] Berard, Ed, "Object-Oriented Domain Analysis," Unpublished, Received through E-mail Jan 92, Berard Software Engineering, Inc, (301)417-9884.

[BERZ89] Berzins, Valdis, "Object-Oriented Techniques Based on Specifications," *SIGSOFT Engineering Notes*, Vol 14(3), May 89, p. 437-438.

[BOBB90] Bobbie, Patrick O.; Urban, Joseph E., "A Knowledge-Driven Methodology for Eliciting and Restructuring Software Requirements for Distributed Design,"

Proceedings of the Second International Conference on Tools for Artificial Intelligence, Nov 90, p. 584-592.

[BOOC91] Booch, Grady, "*Object-Oriented Design with Applications*," c1991, Benjamin/Cummings Publishing Co.

[BULM91] Bulman, David, "Refining Candidate Objects," *Computer Language*, Vol 8, No 1, Jan 91, p. 30-37.

[CARV90] Carver, Doris L.; Cordes, David W., "An Object-Oriented Framework to Support Architectural Design Development," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, Volume 2: Software Track*, Jan 90, p. 349-357.

[CHID91] Chidamber, Shyam; Kemerer, Chris, "Towards a Metric Suite for Object-Oriented Design," *OOPSLA '91*, Nov 91, p. 197-211.

[CHIN89] Chin, David N.; Takea, Koji; Miyamoto, Isao, "Using Natural Language and Stereotypical Knowledge for Acquisition of Software Models," *IEEE International Workshop on Tools for Artificial Intelligence: Architectures, Languages, and Algorithms*, 1989, p. 290-295.

[COLB89] Colbert, Edward, "The Object-Oriented Software Development Method: A Practical Approach to Object-Oriented Development," *Tri-Ada '89*, Oct 89, p. 400-415.

[COLE92] Coleman, Derek; Hayes, Fiona; Bear, Stephan, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design," *IEEE Transactions on Software Engineering*, Vol 18, No 1, Jan 92, p. 9-18.

[COYO91] Coad, Peter and Yourdan, Edward, *Object-Oriented Analysis*, c1991, Prentice-Hall, Inc.

[DAVI90] Davis, Alan M., *Software Requirements Analysis and Specification*, c1990, Prentice-Hall, Inc.

[DOHE90] Doherty, B. S., "Elicitation and Verification of a functional Specification," *ECAI 90, Proceedings of the 9th European Conference on Artificial Intelligence*, p. 234-239.

[FICK88] Fickas, Stephan; Nagarajan, P., "Critiquing Software Specifications," IEEE Software, Vol 5, No 6, Nov 88, p. 37-47.

[FIRE91] Firesmith, Donald, "Structured Analysis and Object-Oriented Development are not Compatible," ACM Ada Letters, Vol XI, No 9, Nov/Dec 91, p. 56-65.

[HAYE91] Hayes, Fiona; Coleman, Derek, "Coherent Models for Object-Oriented Analysis," *OOPSLA '91*, ACM/SIGPLAN, Vol 26, No 11, p. 171-183.

[HOLB90] Holbrook, Hilliard, "A Scenario-Based Methodology for Conducting Requirements Elicitation," *ACM SIGSOFT Software Engineering Notes*, Vol 15, No 1, p. 95-104.

[IPCH91] Ip, Saimond; Cheung, Louis C. Y.; Holden, Tony, "Complex Objects in Knowledge-Based Requirements Engineering," *6th Annual Knowledge-Based Software Engineering Conference*, Sep 91, p. 1-11.

[JAWO90] Jaworski, Allan; LaVallee, David, "Principles for Defining an Object-Oriented Design Decomposition in Ada," *WADAS '90*, Jun 90, p. 173-182.

[JALO89] Jalote, Pankaj, "Functional Refinement and Nested Objects for Object-Oriented Design," *IEEE Transactions on Software Engineering*, Vol 15, No 3, Mar 89, p. 264-270.

[KORS90] Korsen, Tim; McGregor, John D., "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, Vol 33, No 9, Sep 90, p. 40-60

[KUNG89] Kung, C. H., "Conceptual Modeling in the Context of Software Development," IEEE Transactions on Software Engineering, Vol 15, No 10, Oct 89, p. 1176-1187.

[KURT90] Kurtz, Barry D.; Woodfield, Scott N.; Embly, David W., "Object-Oriented Systems Analysis and Specification: A Model Driven Approach," *COMPCON Spring '90*, 26 Feb - 2 Mar 90, p. 328-332.

[LADD90] Ladd, Scott Robert, "Right and Wrong (Picking Classes in Object-Oriented Programming," *Computer Language*, Vol 7, No 4, Apr 90, p. 103-107.

[LADE89] Ladden, Richard M., "A Survey of Issues to be Considered in the Development of an Object-Oriented Development System for Ada," *ACM Ada Letters*, Mar/Apr 89, Vol IX, No 2, p. 78-88.

[LOUC88] Loucopoulos, P.; Layzell, P. J.; Champion, R. E. M.; Gibson, M. D., "A Knowledge-Based Requirements Engineering Environment," *Proceedings of the Conference on Knowledge-Based Software Assistance*, Aug 88, p. 139-154.

[LOUC90] Loucopoulos, P.; Champion, R. E. M., "Concept Acquisition and Analysis for Requirements Specification," Software Engineering Journal, Vol 5, No 2, Mar 90, p. 116-124.

[LUBA86] Lubars, Mitchell D.; Harandi, Mehdi T.," Intelligent Support for Software Specification and Design," *IEEE Expert*, Vol. 1, No. 4, Winter 86, p. 33-42.

[MEYE88] Meyer, Bertrand, "Object-Oriented Software Construction," Prentice-Hall, c1988.

[MONA92] Monarchi, David E.; Puhr, Gretchen I., "A Research Topology for Object-Oriented Analysis and Design," *Communications of the ACM*, Vol. 35, No. 9, Sep 92, p. 35-47.

[MRDA90] Mrdalj, Steven, "Stepwise Object-Oriented System Design," *COMPEURO '90*, May 90, p. 520-521.

[NERS92] Nerson, Jean-Marc, "Applying Object-Oriented Analysis and Design," *Communications of the ACM*, Vol. 35, No. 9, Sep 92, p. 63-74.

[REUB91] Reubenstein, Howard B.; Waters, Richard C., "The Requirements Apprentice: Automated Assistance for Requirements Acquisition," *IEEE Transactions on Software Engineering*, Vol 17, No. 3, Mar 91, p. 226-240.

[ROSS90] Ross, Donald L., "Issues in Object-Oriented Requirements Analysis," *WADAS '90*, Jun 90, p. 77-99.

[RUMB91] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William, "Object-Oriented Modeling and Design," Prentice Hall, c1991.

[RUBI90] Rubin, Kenneth S., "Reuse in Software Engineering: An Object-Oriented Approach," *IEEE COMPCON*, Spring '90, p. 340-346.

[RUBI92] Rubin, Kenneth S.; Goldberg, Adele, "Object Behavior Analysis," *Communications of the ACM*, Vol. 35, No. 9, Sep 92, p. 48-62.

[SAEK89] Saeki, Motoshi; Horai, Hisayuki; Enomoto, Hajime, "Software Development Process from Natural Language Specification," 11th International Conference on Software Engineering, May 89, p. 64-73.

[SCHA92] Schaschinger, Harald, "ESA - An Expert Supported OOA Method and Tool," *ACM SIGSOFT Software Engineering Notes*, vol 17, no 2, Apr 92, p50-56.

[SCHO91] Schoen, Eric, "Active Assistance for Domain Modeling," 6th Annual Knowledge-Based Software Engineering Conference, Sep 91, p. 28-39.

[SHLA89] Shlaer, Sally; Mellor, Stephen J., "An Object-Oriented Approach to Domain Analysis," *ACM SIGSOFT Software Engineering Notes*, Jul 89, p. 66-77.

[SHLA88] Shlaer, Sally; Mellor, Stephen J., "*Object-Oriented Systems Analysis: Modeling the World in Data*", c1988, Prentice-Hall Inc.

[SIBL89] Sibley, Edgar H., "An Object-Oriented Requirements Specification Method," *Communications of the ACM*, May 89, Vol. 32, No. 5, p. 608-623.

[TSAI88] Tsai, Jeffrey J. P.; Ridge, Joel C., "Intelligent Support for Specification Transformations," *IEEE Software*, Vol. 5, No. 6, Nov 88, p. 28-36.

[TSAI89] Tsai, Jeffrey J. P.; Tsai, Shun-Tzu; Liu, Alan, "A Frame and Rule Based System to Support Software Development Using an Integrated Software Engineering Paradigm," *IEEE International Workshop on Tools for Artificial Intelligence: Architectures, Languages and Algorithms*, 1989, p. 282-289.

[TSAI91] Tsai, Jeffrey J. P.; Weigert, Thomas, "HCLIE: a Logic-Based Requirement Language for New Software Engineering Paradigms," *Software Engineering Journal*, Vol 6, No 4, Jul 91, p. 137-151.

[WALT78] Walters, Neal, "An Ada Object-Based Analysis and Design Approach," *Ada Letters*, Jul/Aug 91, Vol XI, No 5, p. 62-78.

[WHIT89] Whitcomb, Mark J.; Clark, Boyd N., "Pragmatic Definition of an Object-Oriented Development Process for Ada," *Tri-Ada '89*, Oct 89, p. 380-399.

[WHIT90] Whiting, Mark, "Workshop: Finding the Object," OOPSLA/ECOOP '90, Oct 90, p. 99-107.

[WINB90] Winblad, Ann L.; Edwards, Samuel D.; King, David R., *Object-Oriented Software*, c1990, Addison-Wesley Publishing Co.

[WIRF90] Wirfs-Brok, Rebecca, "Surveying Current Research in Object-Oriented Design," *Communications of the ACM*, Vol. 33, No. 9, p. 104-123.

[WROB88] Wrobel, Stefan, "Design Goals for Sloppy Modeling Systems," *International Journal of Man-Machine Studies*, Vol. 29, No. 4, Oct 88, p. 461-477.

[YAUL88] Yau, Stephen S.; Liu, Chung-Shyan, "An Approach to Software Requirement Specification," *IEEE COMPSAC '88*, Feb 88, p. 83-88.

[ZERO91] Zeroual, K, "KBRAS: A Knowledge-Based Requirements Acquisition System," *6th Annual Knowledge-Based Software Engineering Conference*, Sep 91, p. 40-52.

REPORT DOCUMENTATION PAGE

	September 1993	Doctoral Dissertation

4. TITLE AND SUBTITLE

On the Automation of Object-Oriented Requirements Analysis

5. FUNDING NUMBERS

6. AUTHOR(S)

Nancy L. Crowley, Major, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/DS/ENG/93-11

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Software Technology for Adaptable Reliable Systems (STARS)
Suite 400
801 North Randolph Street
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT

Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The research investigated the possibility that an object-oriented requirements analysis (OORA) specification model can be represented in a computer system and used as a basis for the elicitation of the information necessary for the development of an object-oriented specification for a particular problem. The proof-of-concept system developed is called the OORA Automated Knowledge System (OAKS). OAKS contains a generic domain model that is modified to satisfy a particular problem in the domain. The core of OAKS is a reusable domain model, which represents a domain of interest. The domain model is used as a basis for user changes that are made to meet specific requirements of a particular problem. The domain model was structured to allow it to be ported to other domains of interest and inserted into the OAKS system. Therefore, OAKS represents an OORA system that can be used in numerous domains to develop an OORA specification for a particular problem.

14. SUBJECT TERMS

Object-Oriented,Requirements,Requirements Analysis

15. NUMBER OF PAGES

279

16. PRICE CODE

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