An Object Oriented Paradigm for Requirements Specifications.

David W. Cordes
Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
An object-oriented paradigm for requirements specifications

Cordes, David W., Ph.D.
The Louisiana State University and Agricultural and Mechanical Col., 1988

Copyright ©1989 by Cordes, David W. All rights reserved.
An Object-Oriented Paradigm
for Requirements Specifications

A Dissertation
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

David W. Cordes

B.S. University of Arkansas, 1982
M.S. Purdue University, 1984

August 1988
Acknowledgements

I would like to thank Professor Doris Carver for all of the support and assistance that she has provided over the past two years. I deeply appreciate her time and efforts. I would also like to thank my wife, Susan, and my parents for their constant support and encouragement throughout my graduate career.
Table of Contents

Chapter 1: Introduction ................................................................. 1

Chapter 2: Requirements Specification Techniques ......................... 7


Chapter 4: A Requirements Document Evaluation Methodology ........... 93

Chapter 5: An Object-Oriented Specification Methodology ................. 116

Chapter 6: Summary ................................................................. 147

Chapter 7: Future Research ......................................................... 154

Bibliography ........................................................................... 160

Appendix: A Sample Execution of the Specification Paradigm ............ 169

Vita .................................................................................. 196
List of Figures

Figure 1.1: The Waterfall Model ................................................................. 3
Figure 1.2: Software Development Life-Cycle Errors ............................... 5
Figure 2.1: SADT Box-and-Arrow Diagram .............................................. 16
Figure 2.2: Traditional Software Development ........................................... 27
Figure 3.1: Sample Requirements Document ............................................ 54
Figure 3.2: Environment Generation ......................................................... 55
Figure 3.3: ATN Transition Arcs ............................................................... 59
Figure 3.4: Sample ATN Model and Code ................................................. 62
Figure 3.5: Parsed Sentence Format .......................................................... 65
Figure 3.6: Word Definition ..................................................................... 66
Figure 3.7: Prepositional Phrase Ambiguity .............................................. 67
Figure 3.8: Sentence Rephrasing ............................................................... 68
Figure 3.9: Sample Parsed Sentence Structure ........................................... 69
Figure 3.10: Event Formats ................................................................. 72
Figure 3.11: Property Formats ................................................................. 73
Figure 3.12: Hierarchy Formats ................................................................. 74
Figure 3.13: Definition Formats ............................................................... 76
Figure 3.14: Restriction Formats ................................................................. 77
Figure 3.15: Traceability Formats ............................................................... 78
Figure 3.16: Conversion Algorithm .............................................................. 80
Figure 3.17: Property Fact Generation ......................................................... 84
Figure 3.18: Event Fact Generation .............................................................. 85
Figure 3.19: Incomplete Fact Generation ...................................................... 86
Figure 3.20: Multiple Fact Generation .......................................................... 87
Figure 3.21: Abstract Modifier Quantification .............................................. 88
Figure 3.22: Inexact Referencing .............................................................. 89
Figure 3.23: Revised Sample Requirements Document .................................. 90

Figure 4.1: Environment Evaluation ............................................................ 96
Figure 4.2: Completeness Checking ............................................................. 98
Figure 4.3: Property Consistency Checking .................................................. 99
Figure 4.4: Hierarchy Consistency Checking ............................................... 100
Figure 4.5: Event Consistency Checking ...................................................... 101
Figure 4.6: Necessity Checking ................................................................. 102
Figure 4.7: Document Report Format ......................................................... 105
Figure 4.8: Sample Document Errors .......................................................... 107
Figure 4.9: Sample Document Warnings ...................................................... 108
Figure 4.10: Sample Document Cross-Reference ....................................... 109
Abstract

Software engineering defines a formalized five-step life-cycle for software development. These steps are: requirements specification, design, implementation, testing and maintenance. The requirements specification phase of the software development life-cycle is responsible for determining the functionality of the proposed system. In this work, a methodology is developed that enhances the generation of accurate requirements specifications, utilizing an object-oriented paradigm.

This research realizes four objectives. First, the process of information transfer between the user and the specification team is enhanced. Second, a working base of knowledge containing the domain-specific information within the initial requirements document is established for use by the specification team. Third, techniques for evaluating the overall quality of the initial requirements document are addressed. Specifically, the problems associated with document ambiguity, completeness, consistency and structure are examined. Finally, a specification paradigm is defined utilizing this knowledge-based specification environment. The paradigm permits the automatic generation of an object-oriented specification model. This model may then be used as an input for the design phase.

This paradigm defines a methodology for the establishment and evaluation of the knowledge-based specification environment. The environment permits the incorporation of an object-oriented development strategy into the specification process. In addition, the concept of information traceability throughout the specification process is enhanced.
Chapter One

Introduction

In the early days of computing, the chief concern when designing computer systems involved the system hardware. There was minimal concern regarding the development of the associated system software used in controlling the system. Today, these factors are essentially reversed. The cost of developing system software is a primary consideration during system development. The ability to create reliable, efficient and cost-effective software in a timely manner often determines the overall success of the entire project. As a result of the growing emphasis placed on the software aspects of the system development process, the discipline of software engineering has emerged within computer science. This discipline provides a formalized approach to software development. Its basic intent is to "establish and use sound engineering principles in order to obtain economical software that is reliable and works efficiently" [Nau69].

The major concepts utilized within the software engineering discipline grew out of the work done in engineering hardware systems. Similar to the approach taken during hardware development, software engineering attempts to define a set of methods, tools and procedures for use during the software development process. Unlike hardware development, however, software development is primarily a creative task. This creativity hinders the establishment of formalized software development processes.
Software engineering defines the software development process in terms of three basic concepts. These three concepts correspond to the definition, development and maintenance of the proposed system. System definition creates a full conceptual description of the proposed system. System development then incorporates this conceptual description into an actual system implementation. Finally, maintenance accounts for necessary system upkeep during its production lifetime.

Today, software engineering practitioners recognize a formalized five-phase software development life-cycle model as the basis for methodical system development. This model, known as the waterfall model, is shown in Figure 1.1. The model provides a logical, structured approach to the problem of developing reliable software. The five phases within this life-cycle model include:

- requirements specifications
- design
- implementation
- testing
- maintenance

Each of these phases has a unique and important role within the overall life-cycle. The functionality of the proposed system, i.e. "what" the system does, is defined within the requirements specifications phase. The determination of exactly "how" to implement this functionality for the proposed system is the responsibility of the design phase. The actual coding of this design is accomplished within the implementation phase. Verification that the code produced during the implementation
phase does indeed match the functionality of the system, as defined during require-
ments specifications, is performed during the testing phase. Finally, the management
and modification of the system during its operational life span is the responsibility of
the maintenance phase.

Figure 1.1: The Waterfall Model
The waterfall model for software development is the most widely accepted method for program development. It does, however, contain a few inherent limitations [Pre87]. The actual software development process does not normally flow through the model as illustrated. Instead, the process iterates through this path, or subsets of this path, during system development. In addition, the waterfall model requires patience on the part of the user, as a working product is not quickly generated. Due to these inherent problems, alternative development techniques have been proposed. Two such techniques are the automatic programming paradigm and a prototype modeling paradigm.

The automatic programming paradigm provides a software development system that automatically translates an initial high-level specification into its equivalent program representation. The major emphasis of such a technique is the development of a correct, reliable and realistic specification. Given this specification as input, the actual system generation processes are completely automated.

The prototyping model for software development recognizes the true iterative nature of this process. It provides a methodology that permits the rapid development of a system model. This model is then built and evaluated. Based on this analysis, another system model is created. This process iterates until the desired model is developed. At this point, a working system is formally developed from the final model. The chief benefit to a prototyping methodology is the improved response time realized during system evaluation.

In spite of the advances made in formalizing the software development lifecycle, software development today remains very much an inexact science. This is
particularly true of the front end of the life-cycle, where a heavy reliance is placed on creativity during the analysis and development of the initial system model. The specification team must essentially "create" the initial system specification, using the initial requirements document as the primary guideline. This inherent creativity often hinders attempts to formalize or structure the process of developing requirements specifications. However, the value of developing a formalized system specification cannot be underestimated. It is upon this foundation that the entire software system is constructed. As shown in Figure 1.2 [Cha86], the major cause of failure within software systems is traced directly to errors occurring within the initial phases of the life-cycle. In addition, it follows that the cost to repair requirements specifications errors is higher than the cost of repairing errors introduced later in the life-cycle. This is due to the fact that any errors introduced during the requirements specification phase require a new iteration through the entire software life-cycle.

![Figure 1.2: Software Development Errors](image)
Given the importance of requirements specifications to the overall life-cycle, improved techniques for their development are beneficial to the total software development process. This research presents a methodology that enhances the development of requirements specifications. It develops an object-oriented specification paradigm. This paradigm relies upon the interactive establishment of a knowledge-based specification environment.

An object-oriented methodology has proven advantageous in other phases of the software life-cycle. The adoption of an object-oriented technique within requirements specifications would realize the same benefits. However, the heuristics associated with both the object-oriented development process and current specification methodologies prevent the direct application of this methodology to the specification process. The paradigm presented in this work defines the mechanisms necessary to incorporate an object-oriented approach to the process of system specification.

This work discusses the development and implementation of this methodology. Its use within and impact upon the requirements specifications phase of the software life-cycle is examined. Existing techniques for the development of requirements specifications are discussed within Chapter Two. A methodology for the generation of a new specification environment is presented in Chapter Three. Techniques for analyzing the information contained within this specification environment are then discussed within Chapter Four. This environment is utilized for the development of an object-oriented specification system, as illustrated within Chapter Five. A summary of the research contributions found within the work is presented in Chapter Six. Finally, future research is outlined within Chapter Seven.
Chapter Two

Requirements Specification Techniques

Requirements specifications represent the first phase in the overall software development life-cycle. The goal of this phase is to determine the true functionality of the system, that is, to determine exactly "what" the functions of the system are. This is generally accomplished through a breakdown and analysis of the English prose requirements document provided by the user. In addition to its responsibility for defining system functionality, the requirements specification methodology is also responsible for determining all necessary constraints imposed upon the system [Kin82]. Constraints include operational, environmental, physical, performance, economic and political restrictions [Rom85]. Operational, environmental, physical and performance constraints are directly related to the stated task. In a less direct manner, economic and political constraints are also related to this task. However, their impact upon the development process must be considered. Inherent system characteristics defined by these constraints can directly influence the specification and design of the system.

In addition to the determination of system functionality and constraints, the task of knowledge transferral occurs within the requirements specification phase. This task is generally not stated as an explicit task within the specification process, but is nevertheless assumed to occur during the development of system specifications. The specification development process is therefore responsible for ensuring that all
relevant problem-specific information possessed by the user is transferred into the software development life-cycle. This transferral process provides the primary link between the user and his/her specific knowledge regarding the application domain and the software development team. As a result, the migration of all relative domain-specific information regarding the proposed system from the user to the system specification team must take place during the specification process.

2.1. Specification Development

A detailed examination of the steps performed during the development of requirements specifications reveals that this process, like any other process, takes a known input, processes this input, and produces a standardized output. Unlike many other processes, however, no one unique technique exists for the process of specification development. A multitude of methodologies exist for guiding the actual conversion of this input into a formalized system specification.

The input to the requirements specification phase is generally presented in the form of an English prose requirements document. This document, generated by the system’s users, contains a user-oriented definition of the system.

Using the requirements document as input, the specification team transforms this document into a set of functional specifications. The specification team extracts all relevant system information from the initial requirements document, and then models the system. The modeling process contains four distinct phases. First, the specific problem presented by the user is identified. This involves a breakdown of the input requirements document into a set of functional requirements relating to the proposed
system. Relevant system characteristics within this document are obtained through an analysis of these requirements. Second, restrictions that could affect system functionality are identified. This involves further analysis of the functional requirements. Specifically, any non-functional requirements of the system, requirements that could restrict system functionality, are identified. Third, a specific specification model is developed. To accomplish this task, an evaluation of all relevant system information is conducted. This set of data, combined with the specification team's own insights and experiences, should prove adequate for a formalized definition of the proposed system. Once this definition is established, a formalized specification model is developed. Finally, a systematic analysis and review of the generated specification is performed.

It is evident, upon examination of the above process, that a heavy burden is placed on the specification team during the development of the system specifications. The guidelines for problem recognition and information evaluation are minimal at best. As a result, the team must rely heavily on heuristic techniques during the functional decomposition of the system. Specification teams are traditionally forced to utilize ad hoc, labor-intensive techniques during their development of system specifications [Sha85].

The output from the requirements specification phase is a formalized specification document. This document is designed for a specific audience, the system design team. The document provides a complete definition of each component within the system. It details the desired system functionality, as required by each of these components. It also describes all possible interactions that can occur between these
components. The output document can be presented in a variety of formats. The two primary formats used for presentation are graphical notations and structured English templates. Regardless of its format, the specification document represents a concrete definition of the nature and functionality of the proposed system.

This point in the life-cycle also marks the end of the user's primary involvement with the software development process. Future user interaction will remain minimal until late in the testing phase. This further illustrates the importance of eliciting all relevant information from the user during specification development.

Realizing that the development of system specifications is not an exact science, software engineers have recursively applied the problem of how to generate proper specifications to itself. Balzer and Goldman [Bal86] have identified the following eight factors as necessary for determining "what" a system specification involves.

First, the specification should be free of implementation details. This step involves system functionality, not system design. Second, an ideal specification should be based on a process-oriented foundation. As most developed systems exhibit a dynamic behavior, a process-oriented model will better illustrate these dynamic characteristics. Third, the specification should encompass the entire system that is being modeled. The proposed system is a collection of interconnected modules. The overall connection and communication structure of this system must also be present within the specification. Fourth, the operating environment for this system should be included as part of the system model. This environment can influence system operation and performance, and should be accounted for during specification. Fifth, the
specification should parallel the real-world model. It should describe the system as the user views it. The objects and events contained within the system should be represented as they exist within the real-world. Sixth, the specification should be based on an operational viewpoint. Since the specification should be sufficiently complete as to be useful in system validation and verification, an operational model that can mimic system behavior is beneficial. Seventh, the specification should be able to account for missing information. The specification must be able to provide useful results, even if it is lacking the complete set of relevant information. It must also be able to incorporate new information as it is introduced. Finally, the specification should be a series of separate, connected modules. Changes or modifications to one element of the specification should not effect the entire document. By minimizing connectivity, the effects of updates and modifications on the specification can be minimized.

These factors capture the characteristics inherent within a properly developed system specification. By adhering to these guidelines for specification development, the specification team is able to improve the overall quality of the specifications that they develop. These guidelines provide a set of criteria by which any generated specification can be evaluated.

2.2. Inherent Specification Problems

A number of inherent problems exist within the requirements specifications phase of the life-cycle. These problems are a direct result of the communication intensive nature of this phase. Ramamoorthy et al. [Ram86] identified two structural
inadequacies of the requirements specification phase. First, very large amounts of information are being manipulated during this phase. Large-scale software engineering projects frequently begin with initial requirements documents consisting of tens to hundreds of pages. Management of the information contained within such a document presents a huge logistical problem. In such a context, vital information is often overlooked, ignored or misplaced. Second, the method for presentation of the initial requirements is generally not precise. Since these requirements are developed by the system’s users, they are normally expressed in a natural language format. Unfortunately, the English language utilized for this task is inherently ambiguous. Furthermore, the user often presents information that is inexact, or fuzzy, in nature. The precise information required for specification development is frequently not present within the document.

These problems, inherent to the specification development process, are apparent when examining program errors. Goodenough and Gerhart [Goo75] examined and classified potential software errors. They found four distinct types of errors that can occur within the software development life-cycle. These are:

- **requirements errors** - the failure to specify a given requirement
- **specification errors** - the failure to satisfy a given specification
- **design errors** - the incorrect design of a given specification
- **construction errors** - the incorrect construction of a given design

Two of these four categories, requirements errors and specification errors, are a direct result of a breakdown within the requirements specification phase of the life-cycle.
Requirements errors are created by a flaw within the process of eliciting information from the user. Specification errors reflect problems within the modeling of the proposed system.

2.3. Existing Specification Techniques

The development of a proper specification model presents the specification team with a very large and laborious task. Methods for the elimination or reduction of work within this phase help to improve the overall effectiveness of the developed specification. A wide range of techniques exist that assist in the generation of requirements specifications. These methods utilize a variety of approaches, and are based on a number of underlying concepts. Section 2.3.1 considers conventional techniques for requirements specification. Section 2.3.2 presents a more formalized approach to the development of requirements specifications. Section 2.3.3 considers an alternative approach, that of object-oriented software development. Section 2.3.4 introduces the application of a knowledge-based approach to the system definition problem. Section 2.3.5 then discusses a number of alternative approaches to the problem.

2.3.1. Conventional Methodologies

Conventional specification methodologies define a specification process that includes techniques for data analysis, problem recognition and the development of the actual specification. Their concern focuses on developing techniques for guiding and organizing the tasks involved within the specification process. These methods define a series of guidelines which enable the specification team to decompose the
requirements document in an ordered and logical manner. Emphasis is placed on providing the specification team with a suitable environment for the organization of their work. These systems generally assume a high competence level on the part of the specification team, relying on the team to develop the proper specification. This specification is assumed to be a correct definitional model of the proposed system.

The majority of requirements specifications systems in existence today fall into this category. The first technique to enjoy widespread acceptance within the area is PSL/PSA [Tei77]. The features and characteristics inherent to PSL/PSA established a basic foundation for conventional specification methodologies. Since then, numerous methodologies have been proposed for the development of requirements specifications. Of these, two systems, SADT [Ros77] and SREM [Alf77], have received widespread recognition and acceptance within the field. An examination of these three systems is now presented.

**PSL/PSA**

Problem Statement Language / Problem Statement Analyzer, PSL/PSA, develops a structured model of the proposed system [Tei77]. The Problem Statement Language, PSL, is used to create this procedural model. The model uses a structured English template and pre-defined *key words* to express the information contained within the initial requirements document. The model provides a formalized definition for all objects within the proposed system model, the properties of these objects, and the relationships that exist between these objects. However, it relies on a heuristic transferral of information from the requirements document into this language format.
No formalized process exists for the generation of the initial PSL document.

The Problem Statement Analyzer, PSA, utilizes a data-base approach for analysis of the PSL document. The PSL document is entered as data into the PSA module. Data-base queries, i.e. the commands governing the analysis process, are then accepted from the user. Command processing provides a method for partial evaluation of the information contained within the PSL document. The analysis routines generate reports that contain the requested information. The evaluation is limited to syntactic matching techniques, such as input/output matching between procedures and the detection of breaks in overall information flow.

As mentioned, PSL/PSA provided a foundation for the development of conventional specification methodologies. The basic concepts inherent to this methodology are realized in the majority of the conventional methodologies in use today. These concepts include the utilization of a structured template to capture the information present within the initial requirements document and analysis techniques for evaluation of the information presented within this document.

**SADT**

The most cited work in the field is the Software Analysis and Design Technique, SADT, developed by Douglas Ross [Ros77, Ros85]. SADT utilizes three major concepts in its approach to requirements specifications. First, it supports a formalized method for system decomposition. Second, it provides a graphical notation for the illustration of this decomposition. The notation is based on a box-and-arrow format, as shown in Figure 2.1. The boxes represent the parts that compose the system, while
the arrows illustrate the relationships that exist between these parts. This box-and-arrow format defines the input to each system module, the output it produces, a mechanism for performing this process, and a control over the process itself.

![SADT Box Diagram](image)

Figure 2.1: SADT Box Diagram

The system specification is defined by a series of diagrams. A diagram is constructed from a series of box-and-arrow diagrams. To aid in diagram comprehension, each diagram is restricted to containing six or fewer distinct pieces. Diagrams requiring a more complex representation must be decomposed into a series of subordinate diagrams. In addition, each diagram is accompanied by a supporting English text that describes the role of the diagram within the system.

The third feature included within SADT is its ability to present multiple viewpoints of the proposed system. SADT advocates two different views within the specification, an activity-based approach using actigrams and a data-oriented approach using datagrams. Box-and-arrow diagrams corresponding to each of these
views are developed in parallel. By providing two alternative viewpoints for system specification, the overall system model is strengthened. SADT relies heavily upon a user-specification team validation cycle for determination of specification correctness. The graphical notation it utilizes enables the user to assist in the verification of the developed specification.

SADT provides two major benefits to the process of specification development. First, it presents a structured development technique for the specification process. The box-and-arrow diagrams provide an excellent mechanism for decomposing, organizing and guiding the specification process. Second, it introduces a communication medium that is both clear and precise. The graphical notation and its accompanying English textual description provide the basis for improved communication during the specification process.

SREM

The other major effort in this field is the Software Requirements Engineering Methodology, SREM, developed by Mack Alford [Alf77, Alf85, Sch85]. SREM utilizes a finite-state machine as its model for specification development. Each state in the model represents a unique phase of execution within the system. Each state has a specific input, output, and an internal function that is executed during transition through the state. The overall finite-state machine operates using a stimulus-response method of behavior.

The basic methodology provides a useful framework for the development of specifications. However, large-scale systems often translate into finite-state machines
that are complex and difficult to comprehend. The notion of an R-net is defined to assist with specification comprehension. An R-net defines a network that encompasses all paths of the finite-state machine that respond to a given stimulus. This technique provides a much needed organizational concept for the SREM methodology.

A formalized specification language, Requirements Statement Language, RSL, is used to define the finite-state machine. A graphical notation also provides an alternative format for the representation for these states. RSL reduces the ambiguity found in a natural language interface. The language defines elements, relationships, attributes and structures. The elements correspond to the actual objects and concepts present within the system. Relationships define a hierarchy for the objects and the dependencies that exist between these objects. Attributes are used to modify or qualify the elements defined in the system. Structures provide a mechanism for illustrating information flow through the system.

SREM also provides a series of tools, referred to as the Requirements Engineering Validation System, REVS, for analysis of the developed specification. These tools provide the capability to check the generated RSL document for completeness, consistency and traceability. In addition, they provide facilities for system documentation and generation of simulation routines. Timing constraints are incorporated into the tool kit through the establishment of validation points within the network. These points represent timing checkpoints for the simulation processes.

The checking performed by REVS is based entirely on their system model. The checks are therefore confined to the context of the developed finite-state machine. Consistency checks only ensure that all data is defined before its use. Completeness
checks only test the possible generation of all desired outputs.

The SREM methodology has been expanded to encompass the realm of requirements analysis. SYSREM establishes a methodology for transition of the functionality defined in the requirements document into a standard RSL template. However, this conversion process is still heuristic. Its effectiveness is determined solely by the expertise of the associated specification team.

SREM realizes two major advantages in its methodology for specification development. First, it provides the necessary facilities for the incorporation of time into the specification process. This proves beneficial to the modeling of real-time or online systems. Second, it provides a formalized model for the underlying processes involved in the system. The utilization of a finite-state machine as its model stresses the generic nature of this model.

These three systems, PSL/PSA, SADT and SREM, capture and illustrate the features inherent in conventional specification methodologies. Such systems generally provide a heuristic front-end to the development process. They define a set of informal yet workable strategies for resolving the problems of large-scale software organization and system information management. Their ability to develop the correct system specification relies to a large extent upon the expertise of the associated specification team. Their main contribution lies in the methodologies they provide for the direction and organization of the actual specification process.
2.3.2. Formal Methodologies

A number of researchers have attempted to provide a solution for the ambiguity present within specifications by adopting a formal specification language. A formal specification language is one with precise, explicitly defined syntax and semantics. The precision involved eliminates the ambiguity within the specifications. Formal specification languages are based on two distinct approaches, operational and definitional. The operational approach provides a methodology for program construction within the specifications. The definitional approach provides for program specification by defining a list of desired system properties, without detailing the actual construction methods.

Formal methodologies exist for both the operational and definitional model. Operational techniques include Gist [Bal83], Gypsy [Goo78], Ina Jo [Sch84], PAIS-Ley [Zav81,Zav82], SPECIAL [Rob77] and Z [Abr80]. Definitional methodologies include ACT-ONE [Ehr85], Clear [Bur81], lota [Nak83] and Larch [Gut85,Win87]. Two sample methodologies, one operational and one definitional, are examined in detail. These samples capture the inherent properties within their respective approaches. Gist [Bal83] illustrates the characteristics present within an operational approach. Larch [Gut85,Win87] provides an example of the basic features inherent to the definitional approach.

Gist: An Operational Approach

Gist [Bal83] realizes an operational approach to the problem of developing formal system specifications. The model developed is essentially a prototype of the
proposed system. The methodology defines techniques for removing all constraints regarding efficiency, method and data from the specification process. By removing efficiency from the specification process, the specification team is free to investigate all possible avenues of program development. Consideration of the relative efficiency of these techniques does not enter into the specification team's decision processes. The elimination of method from the specification process permits the specification team to define the functionality of the system in terms of individual functions. These functions are maintained individually, and not combined into a system model. As this combination of functions determines the actual method for development, maintaining their individual nature removes method from the specification process. The removal of data constraints permits the specification team to operate without consideration of the underlying data representations. A standardized interface is developed that controls all access to data within the proposed system. The specification team only has to consider interaction to this controller.

The specification concentrates specifically on defining sets of acceptable behavior for the proposed system. These sets are based on a finite-machine model, and are represented in the specification by a sequence of states and transitions between these states. The system model consists of demons, types, constraints and relations. The demons within the system represent the individual system processes. Types are used for the definition and storage of data within the system. Relations define the interdependencies that exist within the system. Finally, constraints can be applied to any of these features to limit their scope or impact. The features are defined in a formalized manner, requiring the specification team to possess all relevant information
regarding the proposed feature before the specification is developed.

**Gist** also provides capabilities for validation of the developed specification. Two techniques are presented for this validation. As Gist is an operational methodology, both techniques center on the execution of the specification. The first technique involves the mapping of the developed specification into an equivalent representation that is capable of being executed directly, when provided with a set of user-defined execution guidelines. The second technique uses symbolic execution of the actual specification. This permits random testing for a given specification model. Combined, these two techniques are capable of both extensive testing of a given specification model and random testing of a multitude of specification models.

In summary, Gist defines a formal specification methodology based on an operational view of the system. This technique is able to separate all concerns regarding efficiency, method or data from the actual specification process. By utilizing its operational nature, it is also able to provide techniques for validation of the generated specification.

**Larch: A Definitional Approach**

The Larch methodology, [Gut85, Win87], provides a definitional approach to formalized specification development. It presents a two-phase approach to the development process. The methodology incorporates both a domain-independent *shared language* and a domain-dependent *interface language*. The shared language is used to develop the basic units involved within the system specification. The interface language then provides a mechanism to translate this high-level specification into a
language-dependent specification model.

The shared language provides the specification team with the concept of a trait. Traits are the basic units within the specification process. An object trait uses an explicit syntax to define the operations and constraints that exist for that specific object within the system. Theorems that can be proved for a given trait represent the trait's theory. This theory contains all equations defined for that trait, as well as all theorems that can be derived from these equations using the standard rules of inference. The use of traits encourage the incremental construction of specifications, as new specifications are developed based on existing traits.

The shared language possesses no information regarding the representation of the data structures within the system. It provides no algorithms for the implementation of the system. No routines are provided for object actions, and no error handling routines are identified. It concentrates solely on providing a definitional model of the system's functionality.

The interface language provides a mechanism for the automatic conversion of the language-independent traits into a language-specific specification model. This specification provides a description of both the routines utilized by the system and the data structures required within the system. Routines defined within the interface language contain an identifying header, the traits necessary to define its functionality, and a body containing any constraints placed on these traits. The system's data structures are identified with a unique header. They contain all traits necessary to identify the nature of this information, and present an formal interface to the routines that utilize this information.
Larch provides a system for requirements specification that is based entirely on a logical definition of the traits found within the proposed system. By utilizing a two-phase methodology for formal system development, the Larch methodology is able to isolate the logical definition of system functionality. This permits system functionality to be determined without consideration of the resulting output specification format. This theory provides a basis for complete system definition. This functional definition is mapped into a specific language-dependent format through the use of the interface language.

Advantages of Formal Specification Systems

A number of advantages exist to using a formal approach when developing system specifications [Mey85]. By replacing the natural language text in the developed specification with a more formal presentation method, potential problems are eliminated from the document. These include noise - extraneous information cluttering the document, silence - the lack of information regarding a required component of the system, overspecification - the burdening of the analyst with unnecessary information, contradiction - a lack of consistency within the requirements, ambiguity - a lack of concrete information regarding a particular component, forward reference - the use of items not yet defined within the requirements document and wishful thinking - the inclusion of demands not required by the system.

Despite their advantages, formal specification systems are not adequately suited for use in large-scale software engineering projects [Fin86]. This fault is inherent within the formalism. Such techniques lack the necessary guidance mechanism
required for large-scale system specification. The organizational component required
to manage large-scale software development is missing in a formal approach. Addi­
tionally, such methodologies are generally too far removed from the user’s area of
expertise. The user is unable to effectively interact with the specification team during
specification development.

2.3.3. Object-Oriented Methodologies

An alternative approach to standard specification techniques is that of object­
oriented system development. This approach has its foundation in the work by Parnas
regarding information hiding [Par72], as well as Liskov’s and Guttag’s work on data
abstraction [Gut78,Lis72]. It also relies heavily on the notion of system modularity.
Object-oriented development is fundamentally different from the two standard tech­
niques for program design, functional and data-flow development. In a functional
methodology, the program is decomposed into modules based on functionality, result­
ing in a global data environment. In a data-flow approach, concentration is placed on
the design of the data itself, and its flow through the system.

The object-oriented methodology provides an alternative scheme for the organi­
zation of the system. This approach centers system development around the concept
of an object. All objects in the development process correspond to equivalent real­
world entities. An object’s behavior is characterized its actions within the system.
The object possesses an internal state that is manipulated by both the actions it ini­
tiates and the actions that other objects invoke upon it. The system is organized as a
collection of objects, incorporating ideas from both functional and data-flow
development schemes, as well as utilizing the concepts of data abstraction, information hiding and modularity [Lev84].

The motivation behind an object-oriented approach is easily recognized. An optimal software solution for a given problem should parallel the actual real-world solution techniques. As shown in Figure 2.2 [Led81], standard software development schemes involve both a translation process into the solution space and then back to the problem space. By minimizing the transition between these two spaces, the methodology employed during the development process can better parallel the actual real-world solution techniques. Furthermore, software that closely resembles the corresponding real-world domain aids in the maintenance of the developed system. In comparison to traditional development techniques, an object-oriented approach is capable of providing software that is more easily understood and maintained [Boo86].

The object-oriented methodology is currently evolving along two distinct paths. The programming language and artificial intelligence communities are continuing research into pure object-oriented languages and their programming environments. The software engineering community is investigating applications of an object-based approach to the development processes within the software life-cycle.

Object-Oriented Languages

Object-oriented languages grew out of the work on information hiding, modularity and data abstraction. Its origins are traced back to SIMULA [Dah66]. However, the first language to truly capture the inherent features of an object-oriented world is SMALLTALK [Gol83]. Since then, a number of object-oriented languages have sur-
faced. These include languages such as ACTORS [Lie81], FLAVORS [Wei81] and LOOPS [Bob81]. Additionally, languages such as Ada provide the necessary capabilities to define an object-oriented programming environment. However, Ada does not incorporate these features as inherent characteristics of the language.

While a number of languages exist within the area, they all possess a standard set of features. Using the object for representation of the entities within the system, object communication is performed through the use of messages [Ste85]. These messages are sent by the object that is initiating the action. Messages are assumed to be
initiated independently, as required by the functionality of the objects. A complete series of messages that encompass a specific task or function are combined into a *protocol* for that task. These messages support the concept of data abstraction within the language, as well as providing a natural framework for the implementation of concurrency.

The object-oriented methodology also relies heavily on the concepts of *class* and *inheritance*. A class is a grouping of similar objects. Objects within a class possess *instance variables* relating specific information regarding their individual existence as well as *class variables* detailing information concerning the class to which they belong. Concepts for incorporating both single-parent and multiple-parent hierarchies have been investigated.

A number of benefits are realized through the use of an object-oriented paradigm. First, an object-based language eliminates the need to maintain a data environment during program execution. Conventional programming languages require this environment for type-checking and data-checking. The message-passing concept incorporates all necessary checks regarding the system’s operations [Cox84]. Second, applications such as simulation systems, system programming and artificial intelligence have proved well suited to the characteristics provided by an object-based approach. Third, object-oriented languages are well suited for efforts in concurrent programming. Independent message passing provides a fundamental mechanism for the implementation of concurrency. Finally, unlike other methodologies, the individual objects may initiate actions. This is fundamentally different from conventional passive data-typed languages.
Object-Oriented Software Development

The concept of object-oriented software engineering originated with Booch [Boo82]. Object-oriented development provides a partial coverage of the life-cycle. Its emphasis lies in the design and implementation phases of system development. Object-oriented programming does not provide a formalized methodology for the development of requirements specifications.

The Object-Oriented Development Process

Object-oriented development [Boo82, Boo86, Boo87] presents a technique for the design and implementation of system software. This methodology is based on the work by Abbott [Abb83] regarding informal program design. The object-oriented approach incorporates this work into a formalized paradigm for software development. The methodology presents a five-phase development process. These phases are:

1. Identify the objects within the system
2. Identify the operations performed by the system
3. Establish the visibility of all objects in the system
4. Establish a concrete interface for each object
5. Implement each object

All real-world objects that require modeling within the system are identified during the first phase of the development process. These objects are categorized by their role within the real-world model. Objects are identified as either an actor, an agent or a server. Actors are objects that are capable of invoking independent actions. Servers
represent objects that invoke no actions but are affected by the actions initiated by other objects. Agents represent objects that both suffer and initiate actions. In addition, the grouping of similar objects into classes and hierarchies occurs during this phase. This grouping is based on object functionality. The guidelines presented for recognizing objects within the system utilize a grammatical analysis of the noun phrases within the input document. These guidelines rely on heuristic techniques to identify the objects of interest.

Identification of the system’s operations occurs during the second phase of object-oriented development. The operational characteristics of the objects identified in the first step are formally defined. These characteristics not only identify the operations that an object initiates and the objects affected by these operations, but also include a definition of the constraints that are placed upon these operations. The operations defined for a given object are identified as either constructors, selectors, or iterators. Constructor operations change the current state of the object. Selector operators retrieve the current state of the object. Iterators examine all parts of the object, possibly updating the object’s values during this process. As with the identification of objects, a grammatical analysis of the input document is required for identification of the system’s operations. This analysis locates all unique verbs and predicates within the document. Again, this technique utilizes heuristic techniques during identification of the operations of interest.

During the third phase in object-oriented development, the visibility of each object within the system is established. This permits an analysis of the relationships that exist between the system’s objects. These relationships include hierarchy
relationships, such as inheritance of properties, as well as functional relationships. The definition of object functionality provides a foundation for the determination of system modularization.

A standardized interface for each object is developed within phase four of the process. All interfaces required for access to this object are formally defined. In addition, a module template is generated for each object. This template provides a foundation for the integration of both information hiding and data abstraction into the object-oriented development method.

Finally, implementation of these objects occurs during the last phase of development. No formal methodology is presented for this step. Instead, a standard implementation process is assumed. A recursive application of the five-phase object-oriented development methodology is recommended for objects that are composed of several subordinate objects.

The Ada programming language provides an ideal environment for the development of object-oriented specifications [Boo87, Buz85]. Ada contains mechanisms for parallelism and modularity that are not fully utilized within a standard development technique. The package and task constructs within the Ada language provide a natural interface to the object-oriented development strategy. This integration provides a sound environment for the development of system software.

Enhancements to the Development Process

The object-oriented development paradigm is not a rigorous methodology. It relies heavily on both heuristics and informal techniques during system development.
The methodology is refined [EVB85] into a more formalized development process. This process formalizes a number of the heuristic operations inherent within the original methodology. The original five-phase object-oriented design process is encased within a three-step formalized development procedure. Step One defines the problem. It generates a single-sentence statement of the problem and provides an analysis of the relevant domain information. Step Two then develops an informal development strategy for this problem. The strategy corresponds to the initial input assumed within the traditional object-oriented development methodology. Step Three is equivalent to the original five-phase object-oriented development methodology. The enhanced technique also incorporates a number of software quality assurance checklists into each step of the development process. These checklists are used to help ensure that all necessary tasks are completed during each step of the development process.

The first step of the enhanced methodology attempts to integrate the process of system specification into an object-oriented development scheme; however, it still relies heavily on heuristic decision processes and is primarily designed for small-scale systems. It recommends the use of a formalized specification technique during the initial decomposition of systems of any significant size. The guidance mechanisms necessary for the development of a complete system specification are not present within the object-oriented approach.

**Limitations and Benefits of the Development Process**

The object-oriented paradigm does not provide a formal methodology for requirements specifications. The technique generally utilizes a conventional
requirements specification technique as a front-end to the object-oriented development process. The Jackson System of Development [Jac83], an alternative specification methodology discussed in Section 2.3.5, provides capabilities compatible with the goals of object-oriented development [Boo86]. It provides a clean interface into the front end of the object-oriented development process.

A number of benefits are realized in an object-oriented approach to software development. First, object-oriented development provides an optimal framework for the re-use of software [Mey87]. By developing the system along object guidelines instead of application-specific paths, objects are re-used and re-applied in other applications with a minimum of effort. No longer are these objects custom-tailored to a specific application. Instead, software systems are viewed as a coherent set of related objects. The development team is no longer required to develop each software system independently. Instead, it can cut-and-paste existing objects together as required by the new application. Only the pieces of the proposed system that do not currently exist are constructed.

A second advantage to the object-oriented technique lies in its potential for reducing overall life-cycle costs. Maintenance costs comprise a major portion of the total software cost. By utilizing an object-oriented approach, the resulting code provides a more realistic model of the real-world system. This improved model assists in the reduction of maintenance costs [Buz85]. Additionally, it promotes understandability, thus helping to prevent accidental abuse or error introduction during the maintenance phase.
2.3.4. Knowledge-Based Methodologies

A second alternative approach to the standard techniques for requirements specification has grown out of the field of artificial intelligence. This is the application of knowledge-based methodologies to the problem of generating requirements specifications. The issues of knowledge acquisition and representation within artificial intelligence directly parallel tasks naturally occurring within the realm of requirements specifications. Requirements specifications organize information at a conceptual level, utilizing both function and data abstraction. Knowledge-based applications provide a natural set of data structures to hold this knowledge [Sym88].

Software engineering is, by its very nature, a knowledge intensive task. The strength of a software system often directly parallels the relative amount of domain information present during system development. Information concerning the proposed system is of vital importance to the specification team. This knowledge includes not only information regarding the problem domain, but also relevant information concerning the target machine environment for the system’s implementation. By increasing the amount of domain knowledge present during specification, the potential for development of a more reliable specification is also increased.

Knowledge-based approaches contend that attempts to automate the software development life-cycle should be operational in nature. That is, the product provided should contain information detailing how the computations within the system are actually performed. By utilizing an operational approach, the product is executable, in some manner, from the initial stages of its development. An operational approach assumes that the information generated during a given phase should flow into the next
phase of development with minimal translation. However, the classic waterfall model contains a discontinuity between the "what" and the "how" phases of system development. The system is viewed from a functional standpoint during specification. The next phase of the life-cycle, that of design, requires a change to an operational viewpoint. This logical shift in the method of viewing the proposed system hinders the smooth flow of information between these phases.

A knowledge-based approach to specification realizes two inherent benefits. First, since the initial requirements document is generally presented in an operational fashion, continuing this approach through the requirements specification phase proves advantageous. An operational approach permits the development of a real-world oriented system specification. In addition, it promotes a coherent flow of information from the original requirements document through the specification process and into the design phase. Second, considering the operational nature of a knowledge base, utilizing it as a tool for the representation of the initial requirements document's information provides an improved specification environment. The knowledge base defines a formalized representation for the information contained within the requirements document. It also provides a mechanism for the construction of this knowledge, based on the information contained within the requirements document. Finally, it allows traceability to be incorporated within the specification development process.

Knowledge Base Construction

The generation of a knowledge base for use within software engineering generally involves three distinct steps. First, a methodology for the representation of the
system's domain information is established. Second, methods for acquiring new knowledge are incorporated into the system. Third, a set of tools is provided that enable users to access this contained information.

The first step in knowledge base construction, that of defining a knowledge representation scheme, involves the development of a uniform notation for the storage of knowledge. This notation should be both simple and easily modified. It should be able to support the concept of providing explanations for its responses. Finally, it should be capable of providing a relatively efficient implementation [Wal86].

Four distinct knowledge representation schemes have proven successful in capturing and storing real-world information. These four are: frame-based schemes, semantic nets, rule-based schemes and object-based schemes. Each of these schemes relies on an underlying classification mechanism. The classification scheme generally defines the hierarchy ordering of the objects within the system. This ordering is conveyed through the use of the isa and instance relationships. The ordering also supports the concept of inheritance of properties.

Frame-based techniques provide a formalized template for the storage of information. The frame defines slots for the storage of all information that is relevant to the definition process. Templates are also capable of relating hierarchy information and inheritance properties. A frame-based scheme permits the incorporation of incomplete information into the knowledge base. Missing information is identified by an empty slot marker. The modeling power of frame-based techniques is significant. Frames provide a natural tool for data abstraction. In addition, relations that exist
between these objects can be represented using pointers between frames [Tic87].

Semantic nets, as proposed by Quillian [Qui68], develop an associative network designed to represent the "objective" meaning of the information. The objects within the system are modeled as nodes, and are connected by links that define the relationships existing between these objects. Special links, *isa relations*, permit the incorporation of concepts for object hierarchy and property inheritance.

Rule-based schemes present the knowledge in a series of rules concerning the system's objects. This technique provides a simple method for storage of information. However, such a scheme is hindered by its lack of ability to handle missing information. Contradictions and inconsistencies within the contained information also pose problems for rule-based schemes.

Object-based techniques regard the individual system objects as the foundation for system modeling. These objects can represent either processes or agents in the system, and are characterized by their *state*. An object sends and receives messages with the other objects. These messages invoke actions within this and other objects, possibly changing their states. The benefits to an object-oriented approach have already been discussed.

Pure mathematical logic is not considered as a viable choice for knowledge representation. Mathematical logic lacks the necessary features to deal effectively with abstract concepts. It does not provide any built-in mechanism for the grouping of information, relating property inheritance, or illustrating hierarchy classification.
The second step in knowledge base construction, the act of acquiring new knowledge into the system, plays an important role in the overall effectiveness of the system. A number of methods exist for the acquisition of new information into the knowledge base. These are generally classified into one of three categories [Wal86]. They are: learning by being told, learning by example and learning by observation. The three categories are listed in order of complexity. Learning by being told represents the most basic of the three methods, while learning by observation is considered the most complex.

Learning by being told presents a simple yet powerful technique for acquiring knowledge. Such a method generally relies on a textual input containing the system's actual knowledge. This text is either natural language based or a set of data in a predefined format that corresponds to the knowledge representation scheme being utilized. Techniques exist for both batch processing of the text as well as interactive dialogue with the user. The primary benefit to this approach is the ease of knowledge insertion and the ability to provide explanations for the information contained within the knowledge base. It is weak, however, in compensating for missing information. Systems utilizing this approach must also incorporate techniques for the checking of new information as it is entered into the system. New information is examined to ensure that it is neither inconsistent with existing information nor already deducible from existing information.

Learning by example presents a method that provides a more user-oriented atmosphere for knowledge acquisition. By utilizing a more powerful set of underlying tools, such systems permit the direct acquisition of new knowledge from information
implied during the examination and evaluation of sample data. This technique is motivated by the observation that some expert's knowledge is well-known, but exists in a form that is difficult to explain. By providing a series of examples that demonstrate this knowledge, the system can realize the contained knowledge without requiring the user to express these concepts in concrete terms.

Finally, learning by observation provides an advanced approach to the knowledge acquisition problem. Such a system is primed with a base of knowledge, a method for obtaining additional knowledge, and a guidance mechanism. The system is then allowed to run and accumulate new knowledge independently.

The final step in knowledge base construction is the creation of a set of tools to utilize the information contained within the generated knowledge base. In general, the minimum tool set necessary for a knowledge base must enable the system to solve problems as demanded by the user [Bra86]. This implies the ability of the system to search and analyze the information present in the knowledge base. In addition, methodologies for the retrieval and organization of the information contained within such a system are also provided.

A number of other possible tools can be utilized within a knowledge base. These tools permit improvements to the overall knowledge base environment. First, the system may provide facilities that explain its decisions to the user. Second, the system may incorporate facilities for the handling of uncertain or incomplete information. This generally requires some sort of probabilistic reasoning. Third, an interface may be provided for the user that permits easy access to the information and tools con-
tained within the knowledge base. Such an interface is generally natural language based.

**Knowledge Base Applications**

Knowledge base applications exist for both the design and specification phases of the software development life-cycle. When examining the use of knowledge bases within the design phase of the life-cycle, three systems are considered. These are KBEmacs [Wat86], IDeA [Lub86] and a domain-specific approach [Bar87]. Each of these three systems exemplifies a different approach to the problem of incorporating knowledge into the design process. Such techniques are assisted by the rather structured nature of the design problem.

Requirements specifications present a much more unstable platform for the construction and use of knowledge-based systems. Unless a rigorous environment is established for the development of system specifications, the techniques utilized by standard specification methodologies are difficult to formalize. The creativity employed by heuristic specification algorithms is difficult to capture within a knowledge-based approach. As a result, the use of knowledge-based techniques within this phase of the life-cycle is quite limited. To date, only one major effort exists for capturing the benefits of a knowledge-based technology into the requirements specifications phase. This is the RML system [Bor85, Gre82].

These specification and design methodologies illustrate the characteristics inherent in a knowledge-based approach to software development. Each of these techniques is examined below.
KBEmacs

KBEmacs, a Knowledge Based Editor using emacs [Wat86], provides an intelligent assistant for the task of program design and implementation. The chief concept employed by the system is a cliche. A cliche is defined as a standardized method for dealing with a particular task. The cliche is composed of two distinct parts, a set of roles and an underlying matrix upon which the roles are organized. The roles represent parts of the method that vary between applications. They are well-defined entities, yet are sufficiently flexible as to permit varied applications of the cliche. The matrix represents the relationship between the roles in the cliche, and thus determines the actual functionality of the cliche.

The KBEmacs system provides both high-level and low-level views of these cliches. The cliches are organized into a series of libraries. A common core library is developed, with additional libraries available as required by specific application domains. Full-scale design systems would contain several hundred distinct cliches.

KBEmacs provides an intelligent assistant environment for the development of software. The user selects the desired algorithms for his/her specific task, and the assistant automatically combines these cliches into a single program. The major emphasis of the KBEmacs system lies in the area of software re-use. The same cliches can be applied in a number of distinct applications.

IDeA

IDeA, an Intelligent Design Aid [Lub86], concentrates on the issue of software re-usability. It promotes the concept of a single viewpoint throughout the entire
specification and design process. This viewpoint corresponds to a data-flow oriented model of the system. IDeA provides a series of schema that represent known design components, as well as a set of rules that define a set of standard programming techniques.

The IDeA system takes as its input a high-level data-flow specification. The system then utilizes known schema and programming techniques to automatically construct and refine a software product that corresponds to the given specification. Schema are matched to the specifications provided by the user, and a stepwise refinement is performed upon these schema. High-level schema are decomposed into equivalent low-level schema. IDeA utilizes known programming techniques to link these schema into a coherent module. A dialogue with the user is provided that enables the system to obtain missing information required for system development.

**Domain-Specific Programming**

This knowledge-based technique [Bar87] represents an attempt to produce an automatic programming environment for specific application domains. The system automatically translates a series of formalized specifications and relevant domain-specific background information into a viable program. The system stores the bulk of the domain-specific information as a series of facts and rules within the knowledge base. It also incorporates domain-specific knowledge regarding the types and operations that are characteristic of the objects within this application domain.

In addition, the knowledge base contains a set of pattern-action rules for the actual translation of the formal specification into a corresponding program. These
rules guide the development processes involved in program generation. A matched pattern initiates a set of actions that transform this pattern into a lower-level representation. A series of mathematical translations are also provided, that act upon the formal specification in the case that no pattern-action rules match the current form of the specification.

System operation proceeds by attempting to pattern match the translation rules with the input specification. If no match is found, the mathematical translations modify the form of the specification and try to re-satisfy a match with the translation rules. Once a match is found, the associated actions are executed, and the process is repeated. Successful completion results in the automatic creation of program that corresponds to the initial specification.

**RML**

RML, the Requirements Modeling Language [Bor85, Gre82], develops a system that is designed to capture all relevant real-world knowledge directly into the system specification. These real-world processes are considered essential to the proper modeling of the system. As a result, the specification developed is real-world oriented. It contains all relevant domain objects and events, as well as stating any constraints and assumptions placed on this information.

An object-oriented framework is utilized for this methodology. It is based on the concept of semantic nets, and identifies three distinct items that are to be modeled within the system. These are *object, actions* and *assertions*. Objects model their corresponding real-world entities. These entities can have associated properties.
Additionally, the inheritance concepts of *generalization* and *specialization* are included for the handling of object hierarchies. Actions within the system are modeled as separate entities. The notion of time is supported for these actions, including the definition of *before*, *during*, *after* and *time intervals*. Finally, assertions are included in this framework, defining specific relationships or exceptions to the contained information.

The templates for objects, actions and assertions provide a precise definition for the entity involved. Object templates include components for contained objects, associated variables, object invariants, initial conditions, object producers, object consumers and object modifiers. Action templates include components for action input, output, control, conditions for triggering the action, pre-conditions, post-conditions and other templates involved in the action. Assertion templates identify conditions that must be maintained by the system, as well as constraints upon these assertions.

The RML specification methodology provides a technique for the development of an accurate, time-dependent logical model of the real-world system. RML is considered a knowledge-based specification technique; yet it provides no methodology for the organization of the initial requirements document. As a result, it is suggested that existing specification techniques, such as SADT, be utilized to sort and organize the initial requirements document's information before applying the RML methodology. RML would thus function as a second pass within the specification phase.

Knowledge-based techniques provide a methodology that assists in the process of developing a complete, accurate and consistent model of the corresponding real-
world system. A knowledge-based methodology permits the development of a working model for the system that is defined within the initial requirements document. Such a model is then used as the basis for software development. The operational nature of this technique provides an optimal interface to the initial requirements document, and supports the potential for automatic programming.

A knowledge-based approach provides a simple mechanism for the storage and retrieval of information regarding the application domain. This approach also presents the specification team with a set of tools for examination of the knowledge base's contained information. These tools provide a simple yet powerful method for accessing and analyzing the data. Furthermore, given the operational nature of the domain knowledge, verification of the system specification is accomplished through either trial execution of the generated system model or specification model simulation.

2.3.5. Alternative Methodologies

A number of techniques exist for the development of requirements specifications that cannot be classified as a conventional, formal, object-oriented or knowledge-based approach to the specification problem. These techniques include a variety of alternative paradigms, management-based approaches and database approaches. Two sample approaches are illustrated. First, the JSD [Jac83] methodology is discussed. It defines a real-world based model for system specification. Second, the SPADES [Lud85] specification system is presented. It realizes a database approach to the specification problem.
Among the techniques classified as an alternative paradigm for requirements specifications is the Jackson System of Development, JSD [Jac83]. This technique is designed as a front-end to the Jackson System of Programming, JSP, design methodology [Jac75]. The model specifies the system in terms of its real-world counterpart. The real-world model is decomposed in terms of the entities and actions it possesses. Entities are defined as real-world objects that perform or suffer actions and can be regarded as an individual. Actions represent atomic events that take place in the real-world at a given point in time.

The JSD technique defines a six-step procedure for the development of system specifications. These steps are the entity-action, entity structure, initial model, function, timing and implementation steps. The identification of all relevant real-world entities and actions is performed during the first step within the process, the entity-action step. This is generally accomplished through a grammatical analysis of the initial requirements document. Noun phrases within the document are mapped to system entities, while verbs and predicates are mapped into system actions. A heuristic process is used to determine which entities and actions to select from this document.

A time ordering for the actions defined for a given entity is generated within the second step, the entity structure step. A basic framework is developed for determining possible operation sequences. The framework permits operation sequencing, selection and iteration, yet imposes a strict time-ordering on the relative occurrence of these actions within the system.
The development of a complete system model, in a graphical format, is accomplished within the third step of the process. This model uses a data-flow based approach to represent the interactions between the initial entities within the system. The graphical model developed is also translated into an equivalent textual model. In step four, auxiliary functions are defined that enable the model to produce all of the outputs specified within the requirements document. Any new processes required to generate these outputs are also defined during this step.

Timing is considered in step five. All necessary timing constraints are illustrated for the system model. Finally, the last step is that of implementation. This is actually the design phase of the software life-cycle. The use of JSP during this step is clearly recommended.

**SPADES**

Another popular approach to the problem of developing requirements specifications has evolved from work in the database area. A sample methodology, SPADES, is presented that utilizes this approach.

SPADES [Lud85], a Specification And Design System, is a technique for the development of system specifications that is based on the entity-relationship database model [Che76]. The technique incorporates a methodology for specification, a specification language and a set of tools for evaluating and manipulating the specifications. The objects are formed into classes and relationships and illustrated via links between the objects. The specification is stored in a database. Specific information regarding the specification can be added, deleted or updated as required by the
specification team. A series of tools exist for specification modification. These tools include checkers for consistency and completeness, as well as the ability to graphically display the information contained within the specification.

The developed specification is presented in a structured English framework. This framework formally defines each entity within the specification, all relations involving this entity, and any constraints placed upon the entity.

While this technique provides a system for the management of specification activities, it provides no methodology for the actual specification development process. The technique relies upon the expertise of the specification team to ensure proper system development.

2.4. Summary of Existing Techniques

A number of unique approaches exist for the development of requirements specifications. Conventional techniques, such as PSL/PSA, SADT and SREM provide a complete methodology for the development of the system specification. Such techniques develop a set of guidelines for organizing the specification effort. Formal specification techniques present two distinct approaches to specification development. These are the operational approach and definitional approach. Both methodologies define a rigorous framework for the developed specification, yet offer no guidelines to the overall process of specification generation.

Several alternative approaches exist for specification development. Object-oriented techniques promote the idea of centering system development around the corresponding real-world objects and events, thus providing a realistic model of the
proposed system. Knowledge-based approaches introduce a methodology for the formalized acquisition, representation and reference of the information regarding the proposed system. In addition, other methodologies, such as JSD and SPADES, present unique approaches to the development of requirements specifications.

Each of these techniques contains ideas and concepts that enhance part of the overall specification development process. Each one provides inherent benefits to this process. All of these techniques contribute to the current efforts to formalize the requirements specification process.
Chapter Three

The Generation of a New Specification Environment

Current techniques for the development of requirements specifications encompass a wide range of underlying philosophies, each of which presents a unique approach to the process of specification development. Unfortunately, these methodologies do not yet provide formalized guidelines for the handling of two fundamental aspects within the specification process. First, these methods do not resolve the problems associated with the use of \textit{heuristic} decision processes in the initial stages of specification development. Standard specification techniques rely heavily upon informal analysis techniques on the part of the specification team. Second, methods for the \textit{transferral of domain-specific knowledge} between the user and the specification team are not optimal. Again, standard techniques rely on an informal transition of the data that is deemed relevant by the specification team.

When considering new techniques to improve the processes involved in specification development, emphasis is placed on methods that have demonstrated inherent benefits within other phases of the software life-cycle. An \textit{object-oriented} approach to software development proves advantageous within the realms of design and maintenance. In addition, \textit{knowledge-based} techniques are an effective tool for managing and maintaining information regarding the design phase of the life-cycle. The adoption of both object-oriented and knowledge-based concepts into a requirements specification technique provides significant benefits to the process of
specification development. First, the ability of such a system to utilize an operational approach during the specification process aids in specification validation. An operational model permits the direct execution of the developed specification. The execution output is then validated to ensure that the correct specification was generated. Second, the framework provided by an object-oriented approach minimizes the transitions necessary between the real-world model and the system model. Third, formalized knowledge acquisition techniques decrease the use of heuristic processes during information transferral.

The goal of this research is to develop a new paradigm for the requirements specifications phase of the software development life-cycle. Specifically, to:

Develop an environment that enhances the generation of accurate requirements specifications, utilizing an object-oriented approach.

Four distinct objectives are realized by this research. Each of these individual objectives provides improvements to the general process of requirements specification development. Together, they combine to form a new paradigm for specification development.

The first objective is an enhancement to the process of information transferral between the user and the specification team. The user possesses a great wealth of knowledge regarding the nature of the proposed system. By improving the techniques involved in transferring this information into the specification development process, the overall quality of the resulting specification is also improved.
Second, a working base of knowledge containing this domain-specific information is provided for the specification team. This pool of information is utilized as required during the development of system specifications. The incorporation of a knowledge-based approach provides two distinct benefits. First, the specification team is able to access the user’s domain-specific information regarding the proposed system. Second, it provides the necessary capabilities for maintaining traceability throughout the software development process.

The third objective is an improvement to the quality of the initial English prose requirements document. Existing techniques accept the input document as it is presented. As a result, the specification team must heuristically evaluate and interpret this document before specification can begin. By addressing the problems of document ambiguity, reliability, testability, traceability and correctness, the foundation upon which specifications are developed is strengthened. Ideally, this process should operate interactively, utilizing the user’s domain-specific expertise to resolve these problems. It also provides additional user involvement during the initial phases of the specification process.

The fourth objective of this research is the development of a paradigm for the generation of object-oriented specifications. An implementation of the first three objectives defines a working environment that supports the automatic construction of object-oriented specifications. The representation, function and operations characteristic of the system’s objects, as defined within the initial requirements document, are automatically converted into an object-oriented specification model of the system. This model is then used as an input to the design phase.
An implied ordering exists within these four objectives. The first two objectives, formalized knowledge transferral and representation, generate a basic environment for the development of system specifications. The third objective, an evaluation methodology for the generated knowledge base, defines a user-assisted analysis technique for the contained facts. After the successful generation and analysis of this information, the necessary environment exists for the realization of the fourth objective, an object-oriented specification paradigm.

The generation of the knowledge-based environment is developed within this chapter. A general overview of the process is presented in Section 3.1. The specific steps involved in environment development are presented in Section 3.2 and Section 3.3. Finally, a summary of the knowledge base generation process is presented in Section 3.4. A methodology for the evaluation and analysis of the information contained within this knowledge base is addressed in Chapter 4. An object-oriented paradigm for the development of requirements specifications is then presented in Chapter 5.

To assist in reader comprehension, the processing of a sample requirements document is tracked through this specification process. The information within document is converted into an equivalent set of knowledge base facts. This information is then subjected to a formalized evaluation procedure. After this analysis, an object-oriented specification is generated from the information contained within the generated knowledge base. This document, illustrated in Figure 3.1, is based on an exercise originally presented by Pressman [Pre87]. A detailed example of this paradigm is provided in the Appendix.
All aircraft must have a transponder. The transponder is used to transmit aircraft position to the ground station monitor. The monitor can query an aircraft for information. The monitor keeps a database that maintains this information. A graphics display is generated from the current information. The ground station monitor updates the graphics display frequently. The monitor checks for dangerous situations. The controllers may query the monitor for additional flight information. Controllers may also query the aircraft for this information.

Figure 3.1: Sample Requirements Document

3.1. An Overview of the Environment

An environment is defined that improves the process of information transferral between the initial requirements document and the system specification. Current techniques rely on heuristic conversion processes. This is due to the fact that the initial requirements document often contains inherent ambiguities and partial information. A formalized conversion methodology is defined for the information contained within this document. This domain-specific information is converted into an equivalent set of knowledge base facts. The knowledge-based specification environment is generated through a two-phase conversion of the user's initial English prose requirements document, as illustrated in Figure 3.2.

Parsing of the requirements document is performed during Phase One. This involves an interactive, syntactic parse of the requirements document. A formalized technique is provided within Phase Two for the transferral of this parse information into a format that is accessible to the specification team. A domain-independent knowledge representation scheme is developed for storage of the information contained within the initial requirements document.
3.2. **Phase One: User-Assisted Document Parsing**

The goal of this phase is the parsing of the input requirements document. The methodology employed must possess the capability to translate an input requirements document into a set of parsed sentence structures. The technique relies on human
interaction, in the form of the user who created the initial requirements document, to address the problems of ambiguity within the document.

The input to this phase is a English text representing the initial requirements document. The document is assumed to be grammatically correct. Furthermore, the document must adhere to the following grammatical restrictions. First, no pronouns are permitted within the document. The parsing of pronouns presents no inherent problems, but the associated semantics are not easily resolved. This is due to a lack of information concerning the pronoun's true object of reference. Determining pronoun reference requires a semantic interpretation of both the individual sentence and its surrounding context. Second, imperative and interrogative sentences are regarding as lying outside of the application domain. Questions and commands are not considered part of a standard input requirements document; therefore, such sentence types are not permitted within the document. Finally, restrictions are realized concerning the legal types of noun phrases that can occur within the document. The English language is a nominalizing language. As a result, the grammatical strings that can function as a noun phrase within a sentence are almost unlimited. The parser assumes the use of only standard noun phrases within the requirements document.

Before initiating this parse, a pre-processor is invoked for preliminary manipulation of the requirements document. This processing transforms the data into a format that is recognizable to the parser. Conversions include the translation of the document into lower-case, an elimination of document punctuation, and transformation of all contractions within the document. These transformations permit the simplification of the actual parsing routines.
Parsing proceeds in a sentence-by-sentence fashion. Each sentence is parsed independently. Sentences that do not fit the established guidelines cannot be parsed. These sentences are flagged with an appropriate error message and returned to the user. The user may either re-phrase this sentence in a legal form or drop this sentence from the input document. Legal input sentences are converted into an equivalent parsed sentence structure. The user may also halt the parsing process at any point within the input document.

3.2.1. The Parsing Mechanism

Researchers have devoted many years of study in an effort to develop an automatic parsing technique for the English language [All87,Hei86]. Since natural language parsing is not a trivial task, an attempt to provide a complete solution to the problem of parsing an English requirements document is not feasible for this application. Therefore, a number of assumptions are made regarding the general nature of the parse routine. First, the parser operates in a syntactic manner. It identifies the logical components of each sentence, but does not try to determine the semantic meaning of these parts. Second, the parser's input domain is a subset of the English language. This subset includes only the parts of the language necessary for parsing the requirements document. Third, the parser utilizes an interactive facility to assist in the parsing of inherently ambiguous sentences.

The underlying grammar on which the parser is based is a transformational grammar [Cho59]. The transformational approach to English grammar recognizes a deep, or conceptual, structure for the language. These structures provide a meaningful
representation of the information contained within a sentence. All legal sentences in the English language correspond to one of the basic forms described by these structures. A series of transformations exist that map these structures into surface, or grammatical structures. The surface structures correspond to the actual physical word ordering found within the sentence. Transformations exist for merging sentences, combining subjects or predicates, expressing negation, as well as all other legal English sentence patterns.

The methodology behind the parser is that of an augmented transition network, or ATN [Woo70]. An ATN provides a top-down parse of the input string. It is represented as a finite-state machine, with three important additions. First, the arcs within the network permit transition on both terminal and non-terminal symbols. This introduces the potential for recursion within the ATN. Second, conditional arc transition is allowed. The user may establish a set of conditions to be met that govern the traversal of this arc. The arc is traversed only if these specific conditions are satisfied. Third, the ATN is capable of performing designated actions during arc transition. The ATN maintains a set of user-defined registers that are used to store the information accumulated during the parsing process. These actions manipulate the registers and their contents. As a result, it is possible to retain relevant parse information for later reference. The ATN defined and used within this research is based on the guidelines proposed by both Bates [Bat78] and Finin [Fin83].

A typical ATN system defines a network of states and interconnecting arcs. The network is generally composed of a number of subnets, each of which is designed to recognize and parse a particular component of the overall system grammar. No
required naming conventions exist for the individual states within an ATN. However, a standard convention of "<subnet name> / <last state processed>" is generally adopted.

Control flow within the ATN is accomplished via arc transition. ATNs define a fixed set of transition arcs for this purpose, as illustrated in Figure 3.3. Each element of this set represents one possible method for invoking arc transition. In addition to the transition method, the transition arc also contains an optional user-defined test condition, a possible sequence of actions, and a new destination state. The test condition defines a set of user-defined constraints that help to govern the traversal of this arc. The associated actions permit the retention of information accumulated during arc traversal. Finally, the destination state indicates the new state that is entered upon successful traversal of this arc.

<table>
<thead>
<tr>
<th>Arc Transition Method</th>
<th>Test Condition</th>
<th>Action Sequence</th>
<th>Destination State</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAT &lt;category&gt;</td>
<td>&lt;user test&gt;</td>
<td>&lt;actions&gt;*</td>
<td>(TO &lt;state&gt;)</td>
</tr>
<tr>
<td>WRD &lt;word&gt;</td>
<td>&lt;user test&gt;</td>
<td>&lt;actions&gt;*</td>
<td>(TO &lt;state&gt;)</td>
</tr>
<tr>
<td>MEM &lt;list&gt;</td>
<td>&lt;user test&gt;</td>
<td>&lt;actions&gt;*</td>
<td>(TO &lt;state&gt;)</td>
</tr>
<tr>
<td>JUMP &lt;state&gt;</td>
<td>&lt;user test&gt;</td>
<td>&lt;actions&gt;*</td>
<td></td>
</tr>
<tr>
<td>PUSH &lt;state&gt;</td>
<td>&lt;user test&gt;</td>
<td>&lt;actions&gt;*</td>
<td>(TO &lt;state&gt;)</td>
</tr>
<tr>
<td>POP &lt;form&gt;</td>
<td>&lt;user test&gt;</td>
<td>&lt;actions&gt;*</td>
<td>(TO &lt;state&gt;)</td>
</tr>
<tr>
<td>VIR &lt;type&gt;</td>
<td>&lt;user test&gt;</td>
<td>&lt;actions&gt;*</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: ATN Transition Arcs

The first three arc types, CAT, WRD and MEM, all consume strings from the input. Traversal of a CAT arc indicates that the current input string belongs to the specified category. If the associated user-defined tests are also satisfied, the input is consumed, the specified actions are performed, and the ATN enters the new state
listed. The \textit{WRD} and \textit{MEM} arcs function in a similar fashion, but in these cases the input string must either match the specific word given or be a member of the word list specified. The traversal of a \textit{JUMP} arc provides an unconditional transition to the new state specified, without the consumption of any input.

The \textit{PUSH} and \textit{POP} arcs permit transition between the subnets of an ATN. The \textit{PUSH} arc attempts to successfully traverse the subnet specified. Upon successful traversal of this subnet, the specified actions are performed and control is passed to the new state listed. The ability to traverse an arc on a non-terminal symbol, \textit{i.e.} a subnet within the ATN, introduces the potential for recursion within the parser. The \textit{POP} arc returns control from a subnet to its calling network. The value specified by the \textit{POP} statement is returned to the calling network.

Finally, the \textit{VIR} arc permits the recognition of a previously parsed input string. The values obtained during the parsing of subnets within the ATN can be stored using the \textit{hold} action, and later recognized through a \textit{VIR} arc transition.

Arc transitions may be accompanied by a sequence of actions to be performed by the ATN parser. These actions modify or manipulate a set of registers maintained within the ATN parser. These operations include a means for assigning register values using \textit{setr}, the ability to add new information to register values with \textit{addr} and \textit{addl}, and a method that allows the passing of register values between subnets within the system using \textit{sendr} and \textit{liftr}. In addition, the user may specify his/her own LISP code as actions to be executed during arc traversal.

A sample ATN subnet is illustrated in \textbf{Figure 3.4}. The associated code is also displayed. The "\textit{nounp}" subnet is designed to recognize simplistic noun phrases.
When this subnet is invoked, it enters the "npi" state. While in this state, it matches all input strings that belong to the category det. These are added to the register det. It then performs an unconditional jump to the "np/det" state. In "np/det", it attempts to match input from either the noun or adj category, accumulating this information in the register adj. It can also non-deterministically match a noun as the true noun of the noun phrase, setting the register noun to this value. Once this match is realized, the parser enters the state "np/noun". In this state, prepositional phrases are recognized by a successful push of the subnet "ppi". The information returned by the pop command in the "ppi" subnet is stored in the register mods. The subnet for noun phrases is exited via a pop command, in which the LISP function buildnp generates the output to be returned to the calling network.

The non-deterministic nature of the ATN proves beneficial to the task of parsing the initial document. Through the use of backtracking, the ATN is able to systematically discover all legal parses for a given input sentence. This proves advantageous when dealing with sentence ambiguity, as it illustrates all possible sentence interpretations. The user is then consulted regarding the specific interpretation that is desired for this sentence.

The ATN parser requires access to a dictionary of the English language during its processing. The dictionary defines a vocabulary for the input document. It is referenced during category checking within the arc traversal routines. For each word within the dictionary, the word type, possible word endings, and relevant word features are defined. The word type represents the grammatical class to which the word belongs. These classes recognized by the ATN are adjectives, adverbs,
determinants, modals, negations, nouns, prepositions, pre-verbs, conjunctions, disjunctions and verbs. The word ending list contains all legal endings for the word. This permits the dictionary to store only the base word and its endings. Any number of features can be defined for the word. These features define characteristics of the associated word, such as transitive, intransitive and linking verbs. If a word belongs
to more than one class within the English language, the class, endings, and features fields are repeated for each applicable class.

3.2.2. Incorporating User Interaction

The document parsing phase relies heavily upon user interaction for the resolution of conflicts and ambiguities that arise during processing of the input requirements document. These problems result from the inherent ambiguity within the English language. This ambiguity can be classified as lexical, structural, semantic or pragmatic ambiguity [Kak87]. Lexical ambiguity refers to words within the input document that possess multiple meanings. Structural ambiguity relates to multiple legal interpretations for a given sentence. This is illustrated in the sentence "We saw the man on the road". Was "the man on the road" seen or was the man seen while "on the road"? Semantic ambiguities arise when examining the underlying meaning associated with the sentence. Finally, pragmatic ambiguities arise when consideration is given to the overall context in which the sentence is presented.

The primary role of the user within the parsing process is the resolution of the inherent grammatical ambiguities that are present in the initial requirements document, permitting the parser to function in a purely syntactic manner. The user provides an intelligent source of information for ambiguity resolution. First, the function of prepositional phrases within the input sentences is determined. As illustrated in the example regarding structural ambiguity, prepositional phrases can modify either the nearest noun phrase or the sentence predicate. A distinction between indirect objects and prepositional phrases is also realized. In addition, the function of the *wh-clauses*
that modify noun phrases is determined. Sentences with multiple legal parses are also clarified.

In addition, user interaction is required to maintain an English dictionary for the parser. The English language has a very large vocabulary, with approximately 490,000 general purpose words and an additional 300,000 scientific and technical terms [Gui87]. The average adult with a college education possesses a 10,000 word vocabulary. Given these statistics, the generation of a dictionary encompassing the entire English vocabulary is not feasible. Instead, the parser establishes a core dictionary containing the 1000 most commonly occurring words within the language [Kuc67,Zet78]. The user is responsible for properly defining any words used within the document that do not exist in the parser’s dictionary. An interactive facility is provided for this word definition.

The output generated by this phase is a series of parsed sentence structures. The structures consist of standard LISP S-expressions. The structures contain all relevant information regarding the parsed sentence. The format for the parsed sentence structures is illustrated in Figure 3.5. This information includes the basic sentence type, as well as information regarding the objects and actions present within the sentence. All noun phrases present within the sentence are defined in a nominal phrase structure that contains the associated noun phrase, a determinant list, an adjective list, the object itself, and a list of auxiliary modifiers. These modifiers include both prepositional phrases and wh-clauses. The actions present within the sentence are defined within a predicate structure. This structure contains the name of the associated action, a list of auxiliary verbs, pre-verbs and modals associated with the action, flags that
indicate negation or the use of passive voice, the direct object and indirect object referenced by this action, and a list of modifiers for this action. These modifiers include adverbs, subordinate clauses, prepositional phrases and subordinate conjunctions. The entire sentence also has a set of modifiers that restrict all of the contained information.

```
(sentence (sentence text)
(subject
  (nom { singular/compound }
    ((noun phase text)
      generic/specific
      (noun base noun (actual noun))
      (det (list of determinants))
      (adj (list of adjectives))
      (nounlist (list of nouns))
      (mod (list of modifiers))))
  ( { simple Predicate / compound Predicate }
    (predicate (predicate text)
      { transitive / intransitive / linking }
      (verb base verb (actual verb))
      (aux (list of auxiliary verbs))
      (neg flag indicating negation)
      (passive flag indicating passive tense)
      (direct object ..... → see nom format in subject ← )
      (indirect object ..... → see nom format in subject ← )
      (predicate mods (list of predicate modifiers)))
    (sentence mods (list of sentence modifiers))))
```

**Figure 3.5 :** Parsed Sentence Format

### 3.2.3. Parsing the Sample Document

The parsing process proceeds in a sentence by sentence manner. Sentences that require no user assistance during their parse are displayed as they are parsed. User
assistance is required to define words that do not exist in the parser's dictionary, as illustrated in Figure 3.6 for the sentence "All aircraft must have a transponder". The noun transponder is defined, with the alternative ending "s".

Parsing the sentence (all aircraft must have a transponder)

ATN Error: Undefined word: transponder
Enter word definition, 'help' or 'exit': help

Format for word definitions:

( word type endings features( ) )

Types are noun, verb, adj, adv, pverb, det, conj, sconj, prep, modal, neg
Endings are alternative endings for the word
and include -d -ed -ing -en -ten -s -es -est -st
or '*' for words with no other legal endings
Features for nouns → specific or generic (default)
Features for verbs → trans, linking or intrans (default)
Features for adjective and adverbs → quantifiable
Note: irregular verb forms are expressed as:
   (verbform verb (baseverb (featurelist)) )

Enter word definition, 'help' or 'exit': (transponder noun -s)

Figure 3.6: Word Definition

User interaction is also required to resolve the ambiguity that can arise during processing of prepositional phrases, as illustrated for the sentence "The monitor can query an aircraft for specific information", shown in Figure 3.7. The prepositional phase "for specific information" can modify either the closest noun phrase, "an aircraft", or the sentence predicate, "can query an aircraft". The prepositional phrase is defined as modifying the predicate, explaining what is being queried by the monitor. In addition, the verb "query" is added to the parser's dictionary. All possible verb endings, as well as the type of verb, are indicated within the verb definition.
Parsing the sentence
(the monitor can query an aircraft for specific information)

ATN Error: Undefined word: query
Enter word definition, 'help' or 'exit': (query verb -es-ed-ing features (trans))

What does the prepositional phrase (for specific information) modify?

a. The noun aircraft
b. The predicate itself
c. None of the above

Select 'a', 'b' or 'c' : b

Figure 3.7 : Prepositional Phrase Ambiguity

User assistance is required for sentences that cannot be parsed by the ATN parser. The sentence "The transponder is used to transmit aircraft position to the ground station monitor" uses an infinitive noun phrase "to transmit aircraft position". The use of infinitives is not permitted within the current ATN grammar. The sentence must be re-phrased by the user into an acceptable format, as illustrated in Figure 3.8. Once the sentence is in a form acceptable to the parser, the prepositional phrase ambiguity regarding the phrase "to the ground station monitor" is also resolved.

A parsed sentence structure is generated for each legal sentence within the document. The parse structure for the sentence "All aircraft must have a transponder" is illustrated in Figure 3.9.

3.2.4. Phase One Summary

During the first phase of environment development, a mechanism is constructed for parsing the initial requirements document. The parse mechanism is based on the
Parsing the sentence
(the transponder is used to transmit aircraft position
to the ground station monitor)

Unable to parse sentence
(the transponder is used to transmit aircraft position
to the ground station monitor)

Do you want to:
(a) Re-phrase the sentence and try again
(b) Ignore this sentence and continue parsing
(c) Terminate this session

Select 'a', 'b', or 'c': a

Enter the revised version of the sentence
(the transponder is used to transmit aircraft position
to the ground station monitor)

→ (the transponder transmits aircraft position to the ground station)

What does the prepositional phrase (to the ground station monitor) modify?

a. The noun position
b. The predicate itself
c. None of the above

Select 'a', 'b' or 'c': b

Figure 3.8: Sentence Rephrasing

cancept of an augmented transition network. It provides a syntactic parse of its input.

This mechanism relies heavily upon user interaction for properly resolving ambiguities that exist within the document.

By the end of this phase, the initial English prose requirements document is converted from its textual form into a set of LISP parsed sentence structures. In addition, any grammatical ambiguity that exists within the input document is interactively resolved with the user.
3.3. Phase Two: Knowledge Transferral

The goal of the second phase within the environment generation process is the development of a formalized translation methodology for the information within the requirements document. This data is converted from the parsed sentence structures generated during Phase One into an equivalent knowledge-based representation. First, a standardized knowledge representation format is adopted for storing the user’s
domain-specific information. Second, an algorithm for the conversion of this information into the knowledge-based representation format is presented.

3.3.1. Knowledge Representation Format

The knowledge representation scheme must meet the following criteria. First, the scheme should be sufficiently generic as to permit its use in a variety of applications. The development of complex or domain-specific formats would restrict potential system applications. Second, the scheme must be able to compensate for missing or incomplete information within the initial requirements document. Since the initial document often lacks a portion of the information required for specification development, the representation scheme should permit the inclusion of partial information. However, this missing information must still be identifiable within the generated representation. Third, the scheme should be flexible. The transformational grammar on which English is based permits the generation of a multitude of complex sentence patterns. The representation scheme must be capable of maintaining both the content and the intent of the information presented in such sentences.

Using these general guidelines, a knowledge representation scheme is developed. This scheme is based on the following general observation regarding requirements documents.

Requirements documents express what occurs within the system, what composes the system, and the relationships existing between these system entities.
The information within the initial requirements document is classified as one of three types of data. These types correspond to the events that take place within the system, the properties associated with each of the system's objects, and the hierarchy relationships that exist between these objects. The scheme for modeling this information defines a unique representation format for each of these three data types. The three knowledge representation formats combine to capture all relevant system information presented within the initial requirements document.

These formats contain a set of common characteristics. First, each of the individual event, property and hierarchy relationships that are defined for the system is identified by a unique id. All objects and actions referenced within these facts are also identified with unique ids. The information regarding a single event, property or hierarchy is defined using a series of individual facts that combine to form a frame-based representation of this information.

The knowledge representation scheme utilizes a PROLOG-based syntax. A number of advantages exist to a PROLOG-based format. PROLOG provides a simple, concise mechanism for the generation of facts [Sub85]. It facilitates the development of incremental knowledge, assuming the structure of such knowledge remains consistent. PROLOG also provides built-in features to aid in the process of checking these generated facts.

**Event Representation**

Information regarding the system's events is captured through the format illustrated in Figure 3.10. Each event defined within the system is identified with a unique
The event contains the specific event actor(s) that initiates the event, the action that occurs within the event, the object(s) affected by this event, as well as possible recipient(s), or beneficiaries, of this event. Restrictions can be placed on any of these fields, limiting their scope or range. Restrictions placed on any of the object fields limit the possible set of objects that are involved in this event. Restrictions on the action field constrain the time and manner in which this event may occur. Restrictions are recorded using the restriction fact format, as discussed shortly.

```
EVENT(event_id)
  ACTION(event_id, action_id, restriction_id)
  ACTOR(event_id, object_id, restriction_id)
  OBJECT(event_id, object_id, restriction_id)
  RECIPIENT(event_id, object_id, restriction_id)
```

**Figure 3.10: Event Formats**

**Property Representation**

Information regarding the properties that exist for the objects within the system is illustrated using three distinct *property facts*, as illustrated in **Figure 3.11**. These facts permit the system to capture the object's relevant physical and event-related characteristics, as defined within the requirements document. Physical properties for an object are expressed in two forms. First, simple properties that are characteristic of the object are defined. These properties contain single word modifiers that help to define the associated object. Second, object properties of the object are related. These properties represent features of the object that are defined in terms of other objects.
Event properties relate characteristics of a particular event that the associated object initiates. Event properties define additional information regarding the context in which the associated event occurs. Unless restrictions are placed upon these properties, all three types of property facts are assumed to represent time-independent characteristics of the associated object.

\[
\begin{align*}
\text{property} &\ (object\_id,\ simple,\ property\_id) \\
&\ p\_\text{restrict}(property\_id,\ restriction\_id) \\
&\ p\_\text{simple}(property\_id,\ quantifier\_id,\ value) \\
\text{property} &\ (object\_id,\ object,\ property\_id) \\
&\ p\_\text{restrict}(property\_id,\ restriction\_id) \\
&\ p\_\text{object}(property\_id,\ object\_id,\ restriction\_id) \\
\text{property} &\ (object\_id,\ event,\ property\_id) \\
&\ p\_\text{restrict}(property\_id,\ restriction\_id) \\
&\ p\_\text{event}(property\_id,\ action\_id,\ restriction\_id)
\end{align*}
\]

**Figure 3.11: Property Formats**

**Hierarchy Representation**

The third format required for representation of the information within the requirements document is illustrated in **Figure 3.12**. This format relates information regarding the organizational hierarchy defined for the system’s objects. It permits the establishment of a class-based object environment and provides a mechanism for the development of a hierarchy scheme that includes the notion of property inheritance. In addition, the instantiation of individual objects within a particular class of objects is permitted.
The definition of a system object as a subclass of another class is accomplished with the "isa" fact. The definition of an object as a specific instance of a given class is performed with the "instance" fact. Restrictions upon the individual object that is being defined within the hierarchy relationship are realized through "h_restrict1" restrictions. Restrictions relating to the parent class for this object, as well as any other restrictions that hold for the hierarchy relationship, are defined using the "h_restrict2" restriction.

\[
\text{isa} \left( \text{object id}, \text{class object id}, \text{hierarchy id} \right) \\
\text{h_restrict1} \left( \text{hierarchy id}, \text{restriction id} \right) \\
\text{h_restrict2} \left( \text{hierarchy id}, \text{restriction id} \right) \\
\text{instance} \left( \text{object id}, \text{class object id}, \text{hierarchy id} \right) \\
\text{h_restrict1} \left( \text{hierarchy id}, \text{restriction id} \right) \\
\text{h_restrict2} \left( \text{hierarchy id}, \text{restriction id} \right)
\]

Figure 3.12: Hierarchy Formats

These three formats for the system's event, property and hierarchy facts provide a basic framework that captures the primary information present in the initial requirements document. However, three additional representation formats are necessary to support these generated facts. These formats provide facilities for the definition, restriction and traceability of the information contained within the three primary facts. These auxiliary facts are generated automatically, as required during the production of the basic informational facts.
Definitional Formats

Definitional formats provide a method for the definition of the system's objects, entities and abstract modifiers. These formats are illustrated in Figure 3.13. Each object within the system is defined by its complete noun string, and is identified with a unique object_id. Actions are defined using the associated base verb, and identified through a unique action_id.

Abstract modifiers within the initial document represent a potential source of document ambiguity, thus presenting a problem during document interpretation. Since they are indeed abstract terms, different meanings can be implied by different individuals. Two distinct formats exist for the concrete definition of these abstract modifiers. First, a units-of-measure, along with fixed upper and lower bounds on possible legal values, may be defined for this modifier. This permits the quantification of the original abstract term. Such definitions prove useful during verification of the generated specification. Second, the modifier may be defined using a descriptive English phrase. This phrase relates the implied information that is contained within the original abstract modifier.

A single abstract modifier can imply different meanings at different locations within the document. The system permits this definitional overloading, allowing multiple definitions for a given abstract modifier. During definition of the abstract modifier, all previous definitions for the modifier are presented. The user may either select an existing definition or define a new interpretation for the modifier.
Restriction Formats

Restriction formats provide a method for limiting the scope of the associated information, as illustrated in Figure 3.14. Restriction formats parallel the general structure of property facts. However, while property facts convey time-independent characteristics of the associated object, restrictions only apply to the associated event, property or hierarchy fact. Simple restrictions relate single word entities that constrain the class of objects involved or limit the scope of the associated action. Object restrictions, such as prepositional phrases, define restrictions that are expressed in terms of other system objects. Finally, event restrictions identify actions that impose restrictions on other system facts. All three restriction formats permit the incorporation of nested restrictions. This enables arbitrarily complex sentence patterns to be generated as a set of functionally dependent information.

The restriction "r_0" is used within the generated facts to indicate that no restrictions are placed upon the associated object or action. In a similar manner, the use of the quantifier "q_0" within simple restrictions and simple properties indicates that no
quantification is necessary for the associated modifier.

Traceability Formats

Traceability of the information within the initial requirements document is a major concern during system specification. At each step within the process, the specification team must be able to justify and explain their decision processes. As a result, the information present within the knowledge base is stored in a format that permits its traceability. The traceability formats illustrated in Figure 3.15 permit the knowledge representation scheme to maintain the origin of its contained data. Each individual sentence within the requirements document is stored in the knowledge base. The resulting event, property and hierarchy facts generated from a given sentence are linked to this sentence. Using this scheme, it is possible to trace any of the facts contained within the generated knowledge base to its originating sentence.

The knowledge representation scheme provides a mechanism for storage of the information contained within the initial requirements document. This scheme defines a formalized syntax for the representation of this knowledge. A number of benefits

| restriction (restriction_id, simple, value) |
| r_simple (restriction_id, quantifier_id) |
| restriction (restriction_id, object, name) |
| r_object (restriction_id, object_id, restriction_id) |
| restriction (restriction_id, event, name) |
| r_event (restriction_id, event_id) |

Figure 3.14: Restriction Formats
are realized through such an approach. First, the scheme permits the traceability of its associated information. The origin of all contained knowledge is accessible. Second, the scheme compensates for missing or incomplete knowledge. The generation of incomplete data is permitted. The missing information is determined through an analysis of the generated knowledge base facts, as described in Chapter 4. Third, this knowledge representation scheme provides a formalized representation for the information contained within the initial requirements document.

3.3.2. Knowledge Transferral

Utilizing this knowledge representation scheme, a formalized conversion mechanism is developed that translates the parsed sentence structures generated during Phase One into a set of knowledge base facts. The algorithm functions as a bridge by which the user's domain-specific information is transferred into a knowledge-based specification environment. Determination of the appropriate facts to generate during the conversion process is performed automatically by this algorithm. At the end of this process, a set of knowledge base facts exists that parallels the information contained within the initial requirements document.
The operation of the conversion algorithm, described in Figure 3.16, is based on the sentence’s grammatical properties, as defined in the parsed sentence structure. It converts the parse structure into either event, property or hierarchy information, according to the associated sentence type. These types correspond to three deep structures defined by the transformational grammar, those of transitive, intransitive and linking sentences. The parsing mechanism within Phase One determines the basic sentence type for each of the sentences within the input document. In addition, it maintains information relating the use of passive voice within these sentences. Using this parse data as a guideline, the algorithm generates the appropriate knowledge base facts for this sentence.

Information relating to the events that occur within the system is obtained from two basic types of sentences. First, complete event facts are generated from transitive sentences that do not contain the verb "to have". In such cases, the parsed sentence structure contains a subject, verb, direct object and optional indirect object. This information maps directly into the event knowledge representation format. The sentence subject represents the actor(s) of the event. The action is obtained from the base form of the verb in the sentence. The direct and indirect objects translate directly into object(s) and recipient(s) of the event. Restrictions can be placed on any of these fields, limiting their effective scope.

In addition, incomplete event facts are generated from intransitive, passive sentences within the document, such as the sentence "incomplete event facts are generated from intransitive, passive sentences within the document". The sentence contains an object that is being acted upon, "incomplete event facts", and an action that
for each sentence in the input document do
    check for generation of event facts
    if the sentence type is transitive and
    the verb is not "to have" then
        → generate an event fact
    elseif the sentence type is intransitive and
    the sentence is passive then
        → generate an incomplete event fact (no actor)
    endif
    check for generation of property facts
    if the sentence type is linking and
    sentence complement is not a nominal complement then
        → generate a simple property fact
    elseif the sentence type is transitive and
    sentence verb is "to have" then
        → generate an object property fact
    elseif the sentence type is intransitive and
    the sentence type is passive then
        → generate an event property fact
    endif
    check for generation of hierarchy facts
    if the sentence type is linking and
    the sentence complement is a nominal complement then
        if the subject is a generic entity then
            → generate an isa hierarchy fact
        else
            → generate an instance hierarchy fact
        endif
    endif
od

Figure 3.16: Conversion Algorithm

acts upon it, "generate". However, the actor that performs this action is unknown.

The event is therefore generated with missing information.

Information that defines properties for the objects within the system originates
from three distinct sentence patterns. First, linking-verb sentences without a nominal
complement are translated into simple properties. The subject is mapped to the object that contains this property. The complement represents the simple property defined for this object. Second, transitive sentences utilizing the verb "to have" relate an object property for the sentence's subject. The direct object contains the object property that is defined. Third, non-passive, intransitive sentences define an event property. Such sentences contain an event actor, the subject of the sentence, and an action that is characteristic of that actor. Actual event definition does not occur, as no event object or event recipient is referenced. Instead, a property characteristic of this event is generated.

Information concerning the object hierarchies within the system is obtained from linking verb sentences with a nominal complement. These sentences either relate a subclass to its parent class or define a particular instance of a given class. The appropriate format is determined through an examination of the subject's characteristics, as defined by the features listed for the corresponding noun within the parser's dictionary. The features list defines the noun as either generic, the default, or specific. Sentences involving a generic subject are translated into hierarchy facts using the isa format, while sentences with a specific subject are translated using the instance representation format.

The conversion algorithm requires the ability to determine the specific objects that are being referenced within the parsed sentence structures. This referencing is often hampered by the use of inexact naming and aliasing. As a result, an entity environment is maintained for the definition, storage, and retrieval of the objects.
defined within the system. This environment is responsible for managing the set of objects contained within the initial requirements document. The conversion algorithm invokes the entity environment management routine for determination of the correct object identifier to use during the generation of knowledge base facts. These objects are identified by their complete noun string, as given within the initial requirements document. The environment also maintains a last-use flag for each object. This flag indicates whether or not the object is referenced in the last sentence processed by the conversion algorithm.

Reference to an object in the system can occur through either an exact match to the object name or a partial name match. Partial matching permits the use of inexact referencing within the initial requirements document. This provides the user with greater flexibility during his/her generation of the initial requirements document, as objects such as "the initial requirements document" can be referred to using the simple phrase "document". Inexact objects within the document must be concretely defined by their surrounding context. As a result, inexact matching is only permitted within a limited scope. Inexact matches outside of this range are not guaranteed of mapping to the intended object. The zero-one-infinity principle [Mac83] is used to establish the range of the current context. Specifically, the legal range for inexact matching is one previous sentence. That is, inexact matching is restricted to occurring in only the current sentence and previous sentence within the input document. However, repeated inexact matches may propagate through successives sentences.

The object id to be assigned during generation of the knowledge base facts is determined by the entity environment. A new entity is defined if no exact or inexact
match is located for the associated object name. This new entity is assigned a unique id that is returned to the conversion algorithm for use during fact generation. Objects that generate only an exact match to a known entity have that entity’s id returned for use by the conversion algorithm. Objects that generate only an inexact match, within the allowed context, generally have the corresponding entity’s id returned to the conversion algorithm. However, since the possibility exists that the object referenced could also be defining a new entity that accidentally realizes this partial match, the user is queried to ensure that a partial match is desired and not the declaration of a new object. Finally, objects can provide both an exact and an inexact match within the entity environment. If both matches refer to the same entity, this entity id is returned to the conversion algorithm. If these two matches relate different entities, the user is queried to determine the correct entity referenced within the sentence. This entity’s id is then returned to the conversion algorithm.

Two additional sources of ambiguity are eliminated from the input document during this phase. Both of these are resolved interactively with the user. First, all abstract modifiers within the document are quantified. Using the definitional formats developed for abstract modifiers, a concrete definition of the abstract term is provided. By formalizing its terminology, ambiguity involving the interpretation of these abstract modifiers is eliminated. Second, the functionality of the conjunctions and disjunctions within the initial document is resolved. Conjunctions imply either a grammatical transformation of two sentences with equivalent activities and different objects or the necessity for inclusion of both objects in order to accomplish the associated task. The generated knowledge must distinguish between two objects that are
both capable of performing a given task and two objects that are both required to perform the given task. User interaction is utilized to resolve this ambiguity. Disjunction ambiguity is resolved in a similar fashion.

3.3.3. Fact Generation for the Sample Document

The parsed sentence structures generated from the sample requirements document are converted into a set of knowledge base facts. The conversion process examines the parsed sentence structure to determine the correct type of facts to be generated. The parse information for "All aircraft must have a transponder", described in Figure 3.9, indicates a transitive sentence with the verb "to have". This sentence is converted into the set of facts illustrated in Figure 3.17. The generation of this property data requires the use of two auxiliary formats, the definitional and restriction formats. The objects "aircraft" and "transponder" are defined using the definitional formats, and the restriction "all" is placed upon "aircraft". No quantification is required for the simple restriction "all"; therefore, the null quantifier "q_0" is inserted for this restriction.

| object( o_001 , [aircraft] ). |
| object( o_002 , [transponder] ). |
| restriction( r_001 , simple , all ). |
| r_simple( r_001 , q_0 ). |
| property( o_001 , object , p_001 ). |
| p_restrict( p_001 , r_001 ). |
| p_object( p_001 , o_002 , r_0 ). |

Figure 3.17: Property Fact Generation
Event facts are generated from transitive sentences in the document that do not use the verb "to have". The sentence "The transponder transmits aircraft position to the ground station monitor" generates the set of facts illustrated in Figure 3.18. The object restriction on the action "transmit" indicates where "aircraft position" is transmitted. In addition, fact traceability is realized through the traceability formats. These traceability facts provide a link from the event "e_001" to the sentence that generated this information, "s_002".

```prolog
sentence( s_002, [the,transponder,transmits,aircraft,position,to,the,ground,station,monitor] ).
object( o_002 , [transponder] ).
object( o_003 , [aircraft,position] ).
action( a_001 , [transmit] ).
object( o_004 , [ground,station,monitor] ).
restriction( r_002 , object , to ).
r_object( r_002 , o_004 , r_0 ).
event( e_001 ).
action( e_001 , a_001 , r_002 ).
actor( e_001 , o_002 , r_0 ).
object( e_001 , o_003 , r_0 ).
generator( e_001 , s_002 ).
```

**Figure 3.18: Event Fact Generation**

Fact generation is possible for sentences that do not contain complete information, as illustrated in Figure 3.19. The sentence "A graphics display is generated from the current information" contains no initiating actor. The event data produced by this sentence is generated with a missing actor, "o_0".

Multiple fact generation from a single sentence is also possible, as illustrated in Figure 3.20. The sentence "The monitor keeps a database that maintains this
Figure 3.19: Incomplete Fact Generation

"information" contains a nested event that functions as a restriction on the object "database". Two distinct events are generated from this sentence. First, event "e_003", is generated from the wh-clause "that maintains this information" for the direct object "database". Second, event "e_004" is generated from the primary sentence information, "the monitor keeps a database". A restriction is placed on the object of this event, "database". This restriction relates to the event defined within the wh-clause modifier.

The resolution of abstract modifiers also occurs during the processing of the conversion algorithm. As the conversion process takes place in a sentence-by-sentence manner, this quantification occurs immediately following any ambiguity resolution required by the sentence, as illustrated in Figure 3.21 for the sentence "The ground station monitor updates the graphics display frequently". The word
"frequently" represents an abstract modifier. It is defined by the user as occurring every "10 to 12 seconds". In addition, an abstract modifier may be expressed using a descriptive English phrase. This is also illustrated in Figure 3.21 for the sentence "The monitor checks for dangerous situations". Here "dangerous" is defined to mean having "two or more aircraft in the same air space".

Inexact referencing within the document poses a problem during fact generation. This ambiguity is realized within the sentence "Controllers may also query the aircraft for this information", illustrated in Figure 3.22. Here the reference to "information" may be an inexact reference to the object "flight information" in the last sentence or an exact match to the object "information" referenced earlier in the document. This also illustrates a potential problem within the document, as "information"
The ground station monitor updates the graphics display frequently

The modifier 'frequently' is inherently ambiguous
No definition currently exists for frequently
Building new definition for the modifier 'frequently'

The definition may be in one of the two following two forms
  a) units of measure / upper bound / lower bound
  b) an English description of the modifier

For English description, specify the units of measure as 'English'

In what units is the modifier measured? **seconds**
Are these units ('seconds') specified in:
  (a) integer notation
  (b) decimal notation
  (c) character strings

Please select one of 'a' 'b' 'c' : **a**
If an upper or lower limit is not appropriate, specify 'n/a'
Specify an upper bound for this modifier (in seconds) : **12**
Specify a lower bound for this modifier (in seconds) : **10**

Is the following information correct for 'frequently'? 
→ units: seconds (INTEGER) upper bound: **12** lower bound: **10**
Please select one of 'y' 'n' : **y**

The monitor checks for dangerous situations

The modifier 'dangerous' is inherently ambiguous
No definition currently exists for dangerous
Building new definition for the modifier 'dangerous'

The definition may be in one of the two following two forms
  a) units of measure / upper bound / lower bound
  b) an English description of the modifier

For English description, specify the units of measure as 'English'

In what units is the modifier measured? **English**
Give an precise English description encased in a set of parenthesis
→ (two or more aircraft in the same air space)

Is the following information correct for 'dangerous'? 
→ (two or more aircraft in the same air space)
Please select one of 'y' 'n' : **y**

Figure 3.21: Abstract Modifier Quantification
and "flight information" represent two distinct names for a single system object. This aliasing must be eliminated from the requirements document. In addition, the reference to the object "monitor" in the sentence "The monitor can query an aircraft for information" could either be an inexact reference to the object "ground station monitor" from the previous sentence or the definition of a new object. User interaction is required to resolve this ambiguity. This is also illustrated in Figure 3.22.

<table>
<thead>
<tr>
<th>Controllers may also query the aircraft for this information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two unique objects match the noun in (additional information)</td>
</tr>
<tr>
<td>(a) an inexact reference to the noun 'flight information'</td>
</tr>
<tr>
<td>(b) a specific reference to the noun 'information'</td>
</tr>
<tr>
<td>Please select one of 'a' 'b' : a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The monitor can query an aircraft for information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is the use of (monitor) an inexact reference to the object (ground station monitor) ?</td>
</tr>
<tr>
<td>Please select one of 'y' 'n' : y</td>
</tr>
</tbody>
</table>

Figure 3.22: Inexact Referencing

The environment generation process illustrates several inadequacies within the initial requirements document. All non-standard sentences within the document are replaced with legal input sentences. The abstract modifiers within the document are quantified. The aliasing that occurred within the document is also removed. The resulting requirements document is shown in Figure 3.23.

3.3.4. Phase Two Summary

The second phase in the environment generation process defines a formalized mechanism for the transferral of the user's domain-specific information into the
All aircraft must have a transponder. The transponder transmits aircraft position to the ground station monitor. The monitor can query an aircraft for flight information. The monitor keeps a database that maintains this information. A graphics display is generated from the current information. The ground station monitor updates the graphics display frequently. The monitor checks for dangerous situations. The controllers may query the monitor for additional flight information. Controllers may also query the aircraft for this information.

Figure 3.23: Revised Sample Requirements Document

software development process. This technique requires the development of a standardized scheme for the representation of this information. In addition, an algorithm is developed for the conversion of the parsed sentence structures into a set of knowledge base facts that utilize this representation scheme.

The methodology provides three basic formats for the representation of this information. The data within the initial requirements document is mapped into system events, object properties or object hierarchies. These three formats provide a foundation for the storage of the initial requirements document’s contained information. Three additional formats, definitional, restriction and traceability formats, are provided to fill in the associated details.

The conversion algorithm is driven by the information present within the parsed sentence structures generated during Phase One. Based on the sentence type, the algorithm converts the parsed sentence structures into a set of knowledge-based facts. These facts contain information that is equivalent to the data within the original requirements document. The algorithm relies upon the establishment of an entity environment for assisting in object assignment during fact generation. This
environment ensures that aliasing and inexact naming within the initial requirements
document does not impede the conversion process.

It should be realized that, for a given parsed sentence structure, multiple sets of
information may be developed. When considering all but the simplest of sentences,
the conversion process often involves the generation of multiple facts. This poses no
problems for the conversion algorithm. All information present within a given sen­tence is converted into the appropriate set of knowledge base facts.

3.4. Environment Generation Summary

A technique for the generation of a knowledge-based specification environment
is presented. This process uses the initial requirements document as its input, and pro­duces a set of knowledge base facts. At the end of this generation process, the infor­mation contained within the initial requirements document now exists as an equivalent
set of facts.

Environment generation is a two-phase process. The first task is the conversion
of the initial requirements document into a set of parsed sentence structures. User
interaction is utilized to resolve any inherent ambiguity within this document. The
second task is the conversion of these parsed sentence structures into a set of
knowledge base facts. A representation scheme is defined for this information, and a
conversion algorithm is developed. Two additional sources of ambiguity within the
document are eliminated during this process. A semantic interpretation is provided
for the conjunctions and disjunctions within the document. A method for the
quantification of all abstract modifiers used within the document is also defined.
The generation of an knowledge-based specification environment realizes two of the four major objectives stated for this research. First, it employs a user-assisted technique for the decomposition of the initial requirements document. The process of information transferral between the user and the specification team is enhanced. The user is an active participant in the generation of this knowledge base. Second, a working base of knowledge containing the user's domain-specific information is established. The information now exists in a format that can be utilized during the specification development process. Traceability of the contained information is also possible.
Chapter Four

A Requirements Document Evaluation Methodology

The third objective of this research is the development of a methodology for evaluation of the initial requirements document. Direct analysis of this document is not possible, as the document does not exist in a form that permits automated analysis. However, after the document's information is coverted into an equivalent set of knowledge base facts, the data exists in a format that permits its evaluation. By analyzing this information before it is utilized within a formalized specification methodology, the specification team is provided with a stable foundation for the specification development process.

Prior to the actual evaluation process, additional information is generated from the facts within the existing knowledge base. First, the logical objects involved in the system's events, properties and hierarchies are recognized. During knowledge base generation, the system's objects are viewed in a strictly syntactic manner. The generated objects within the knowledge base facts represent the grammatical subjects, complements, direct objects and indirect objects from the original document's sentences. However, document analysis is best performed using the logical entities involved in the system's events, properties and hierarchies. The logical English components of these facts contain the complete logical entity, such as declaring "components of these facts" to be the logical subject of this sentence. By providing an analysis of this document that offers a logical view of the contained entities, a more
realistic interpretation of the system is presented. The system’s logical objects are
generated automatically, using the information present within the restrictions placed
upon the system’s syntactic objects. Since all of the information required to generate
these logical objects is present within the existing knowledge base facts, this process
is a relatively straightforward transformation of the original factual information.

In addition to the establishment of the system’s logical objects, the visibility of
the logical objects is defined. The visibility of an object includes the set of all other
entities within the system that are event objects or event recipients for events that this
object initiates. Object properties for a given object are also visible to that object, and
thus included in the visibility set. An object’s visibility represents the set of all other
system objects that are required for the definition of this object.

4.1. Evaluation Procedure

The information present within the knowledge base is evaluated before the initia-
tion of a formalized specification methodology. Without such an analysis technique,
errors within the set of domain facts would propagate into the actual specification pro-
cess [Som85]. The information within a proper requirements document should be
complete, consistent, unambiguous, verifiable, traceable, necessary and modifiable
[Cha86]. The document analysis methodology must be capable of addressing these
characteristics for the initial requirements document. Additional traits can be defined
for this document, including correctness, usability, efficiency, realism, transportability
and maintainability. Since these traits are more subjective in their nature than those in
the initial list, they are normally determined informally by the individual user or
specification team. They are not addressed during document analysis.

This research presents an analysis technique, shown in Figure 4.1, that is designed to analyze the reliability, testability and traceability of the initial input requirements document. Coupled with the resolution of document ambiguity during generation of the knowledge base facts, these checks encompass the criteria established for a proper requirements document. Two outputs are generated through this evaluation process, a document report and a regenerated requirements document. After successful evaluation of this set of knowledge base facts, an environment exists that permits the establishment of an object-oriented specification technique.

The evaluation methodology utilizes both stand-alone system checking and user-assisted analysis. The checks are performed automatically by the system, with the exception of object property consistency checks. The user is also expected to provide an overall evaluation regarding the correctness of the document's information.

4.1.1. Reliability

Reliability checking involves an evaluation of the document's information for completeness, consistency and necessity. The assurance that these three properties are satisfied for the document, along with the notion of document correctness, ensures the overall reliability of this data [Adr86].

Completeness

To be considered complete, the document must exhibit three fundamental characteristics. First, no information is left unstated, or "to be determined". Second,
Figure 4.1: Environment Evaluation

the information does not contain any undefined objects or entities. Third, no information is missing from this document [Boe84]. The first two properties imply a closure of the existing information, and are commonly referred to as internal completeness. Internal completeness ensures that all of the information present in the document is
completely defined. The third property relates to the external completeness of the document. External completeness ensures that all of the information required for problem definition is found within the document. This idea is closely related to the concept of correctness, and requires a subjective analysis of the document.

The document analysis routines evaluate the internal completeness of the information contained within the initial document. All missing or incomplete information is identified. Four different checks are performed on the knowledge base facts during completeness checking. The first two checks evaluate the object definitions provided within the document. The first ensures that a minimal set of properties exists for each object. The second tests for the existence of hierarchy relationships that define the relative position of this object within the overall system hierarchy.

The third completeness check ensures that the events described within the system are completely defined. It examines each of the system's events for missing information. Events lacking basic information, such as a missing event actor, are exposed during this test. The fourth check ensures that all events referenced within the system are defined. Events that are referenced through an event property definition must also have the associated event defined. The algorithm for evaluating completeness is described in Figure 4.2. The overall correctness and external completeness of this information must be considered separately by either the user or the specification team.

Consistency

Document consistency implies a lack of contradiction within the information presented. Two types of contradiction exist, either a direct refutation of previously
Check object information
   find all logical objects
   that exist within the system
   for each logical object do
     find all properties
     that exist for this object
     if no properties are located then
       → flag potential error - no defining properties for this object
     endif
   find all hierarchy relationships
   that exist for this object
   if no hierarchy relationships are located then
     → flag potential error - no enclosing hierarchy for this object
   endif
od
Check event information
   find all events
   that are defined within the system
   for each event located do
     if the actor of the event is "o_0" then
       → error - missing actor for event
   endif
od
Check event definition
   find all the event properties
   that are defined within the system
   for each event property located do
     determine the actor that initiates this event
     determine the action that transpires this event
     if no event exists with this actor/action pair do
       → error - no corresponding event for a given event property
     endif
   od

Figure 4.2: Completeness Checking

stated data or an indirect denial of this information. Direct refutations represent incompatible statements within the document. The truth of the first statement directly negates the truth of the second statement. Indirect refutation presents a much more subtle contradiction. A given set of facts could establish a potential situation that,
given the proper set of circumstances, would contradict other facts within the knowledge base.

Determining document consistency is primarily a semantic task. Six distinct consistency tests are defined within the evaluation methodology. Five of these tasks are performed independently by the system. However, it is necessary to utilize user interaction during the evaluation of object property consistency, as illustrated in Figure 4.3. The system identifies all known properties for a given object, presenting them to the user for evaluation. Information deemed inconsistent is flagged with an appropriate error message in the document report.

```
Check system properties
  for each logical object in the system do
    find all properties
    that are defined for that object
    display these property for user evaluation
    if the properties are deemed inconsistent or inadequate by the user then
      → error - inconsistent object properties defined
    endif
  od
```

**Figure 4.3 : Property Consistency Checking**

The next three tests, illustrated in Figure 4.4, ensure that the hierarchy established within the knowledge base is consistent. They check that each *generic* object in the system has at most one class defined as its parent class, and that each *specific* object in the system is an instance of at most one distinct class. The tests also ensure that no circular class definitions exist within the system.

Finally, the last two tests, illustrated in Figure 4.5, evaluate consistency between the events that are described within the system. The events defined within the system
correspond to a set of procedures within the real-world model. The functionality of these procedures is assumed to remain constant. The events defined within the knowledge base should also maintain this consistency. Therefore, a procedure defined by a given event actor and event action should always act upon a consistent event object. Multiple events with the same actor and action are examined to ensure a common object for this procedure. Similarly, procedures defined by a given event action and event object should always be invoked by a constant event actor. Events containing the same action and object are checked to ensure a common actor within this procedure.
Check system events

**find** all unique *actor/action* pairs
that exist within the event definitions

**for** each pair found **do**
**find** all events in the system
that match this actor and action
if different objects are being acted upon in these events **then**
→ flag potential error - inconsistent event objects
**endif**
**od**

**find** all unique *action/object* pairs
that exist within the event definitions

**for** each pair found **do**
**find** all events in the system
that match this action and object
if different actors instigate these events **then**
→ flag potential error - inconsistent event actors
**endif**
**od**

**Figure 4.5 : Event Consistency Checking**

**Necessity**

The requirements document should not contain any information that is unnecessary for solution development. Such information represents noise that must be filtered out during the identification of relevant specification information. Necessity checking of the knowledge base facts overlaps with the process of completeness checking. A true distinction between the processes involved in necessity testing and completeness testing requires knowledge regarding the actual implementation of the system. Only the knowledge obtained from the generated system can determine if the inclusion of a piece of information is needed for specification completeness or if this information is unnecessary to the system’s development.
As a result of this overlap with consistency checking, most of the tests regarding necessity checking are already covered. One additional check does exist, as illustrated in Figure 4.6. This check tests for objects that do not participate in any of the system’s actions. Objects with little or no interaction to other system elements are identified as potentially unnecessary.

```
Check information necessity
   for each logical object defined within the system do
      find all events in the system
         that contain the object as an actor, object, or recipient
      if no events are found that match this constraint then
         -> flag potential error - unnecessary system object
      endif
   od

Figure 4.6: Necessity Checking
```

4.1.2. Testability

Testability implies that the system’s information is presented in a structured, concise, self-descriptive and quantifiable document. Ensuring document testability is accomplished through a two-step procedure. First, the generated knowledge base facts are used to create a new, structured, concise, self-descriptive requirements document. Second, the document’s information is quantified. The knowledge base generation process developed in Chapter 3 requires the quantification of all abstract terms within the document. Ambiguities resulting from the use of abstract terminology are eliminated. This quantification process also provides a set of guidelines for test data generation during both the verification of the developed specification and the testing phase of the software life-cycle.
4.1.3. Traceability

Traceability within the document ensures that the source of the knowledge base's contained information is identifiable. This traceability defines a "chain of accountability" within the development process. All information contained within the knowledge base is linked directly to its generating statements within the original document. Likewise, this traceability can identify the specific facts generated from a given sentence. Both forward and backward accounting for the information within this knowledge base is possible.

In addition to the traceability features within the knowledge base, information is provided that can be used by manual tracing techniques. Specifically, a cross-reference of the information found within the initial requirements document is generated. This cross-reference is constructed by sentence, by object and by action. The three cross-references prove useful during modification of the initial requirements document. They both simplify the locating of data that is being updated and provide assurances that all necessary data is modified.

4.2. Document Evaluation Outputs

The document analysis routine produces two summarizing reports regarding the information and errors found within the initial requirements document. The first report is a listing of the initial document, the errors found within this document during the evaluation process, and the complete set of document cross-references. The second document is the regenerated requirements document produced during the evaluation of testability.
4.2.1. Document Report

The document report contains a complete diagnostic output of the analysis routines. The report content parallels that of a compiled program listing, and is described in Figure 4.7. The primary difference is that the input represents an initial requirements document, and not the code for a program. First, a listing of the initial document is provided. This listing identifies each sentence within the document. Second, the complete list of error and warning messages generated during the analysis routines is produced. Error messages indicate known inadequacies within the document. All errors should be corrected by the user prior to formalized specification development. Warning messages relate characteristics of the document that are potentially incorrect. The user must evaluate these warning messages to determine whether or not corrective action should be taken. Third, the report contains exhaustive cross-references of the information present within this document. A cross-reference is provided by sentence, object and action. The sentence cross-reference illustrates all objects and actions present in the sentence, along with their functionality. The object cross-reference provides a listing of all uses and references of this object within the initial document. A cross-reference is also provided for all actions within the system, detailing their location within the initial document.

The document report is intended for examination by the user who created the initial requirements document. The error and warning messages generated within this report are examined and corrected. In addition, the information contained within this report is analyzed for overall correctness. The corrected document is then resubmitted to the environment generation facilities. It is reparsed, new information is generated,
The initial document, listed by sentence
A listing of all known errors within the document
incomplete information
inconsistent information
unnecessary information
A listing of all potential errors within the document
Cross-reference of the document
by sentence
  all objects referenced and their role
  all actions referenced
by object
  all events that involve it
  all attributes defined for it
  all known hierarchy information
  the objects visibility
by action
  all sentences that reference it

Figure 4.7 : Document Report Format

and this new information is then evaluated.

By providing an iterative process for document evaluation, the user is able to
develop a correct, unambiguous, reliable, testable and traceable requirements docu­
ment for submission to the actual specification process. By eliminating these errors
before the actual specification process is initiated, the specification team is presented
with an optimal input for specification development. In addition, an established role
is defined for the user within the specification environment development process.

4.2.2. Document Regeneration

The document analysis process also constructs an alternative version of the ini­
tial requirements document. This document helps to ensure the testability of the infor­
mation within the initial document. It provides a detailed listing of the objects, events and hierarchies that exist within the system.

This document is composed of three distinct parts. The first part contains a listing of the initial document. This provides the user with a point of reference for questions that might arise during examination of the regenerated document. The second part of this document lists the logical objects defined within the system. These are the objects whose events, properties and relationships define the real-world model for the proposed system.

The third part of this document provides a description of the functionality of the system’s logical objects. This description contains a complete listing of all known event, property and hierarchy information regarding the object. First, a listing of all known hierarchy information for the given object is presented. If the object is a generic entity, all subclasses and parent classes for this object are identified. If the object is a specific instance of some class, the class to which this object belongs is identified. Second, a summary of the associated object properties, both physical and event, is provided. In addition, it notes any functioning of this object as an object property for other system objects. Third, a listing is produced that illustrates all of the events involving this object. The event listing is presented in three parts, the events that this object initiates, the events that affect this object, and the events that benefit this object. Finally, a listing is generated of the sentences in the initial requirements document that reference this object. This illustrates the source of the presented information.

Two benefits are realized through the creation of the regenerated requirements document. First, it permits the user an alternative text for evaluation of the
information presented within the initial document. This simple technique provides a powerful tool for the recognition of inadequate or incorrect requirements [Boe84]. Second, this information provides a structured document for use by the specification team during actual specification development. All known information regarding the system's objects is presented in an structured, concise manner.

4.3. Evaluation of the Sample Requirements Document

The sample requirements document, as it exists in Figure 3.23, is evaluated. The diagnostic messages produced within the document report are divided into two categories, known document errors and warning messages indicating potential document inadequacies. The three errors located within the document are illustrated in Figure 4.8. First, the user has declared, through the interactive consistency checks, that the property information existing for the object "aircraft" is inadequate. Second, the actor for sentence s_005 in the document, as illustrated in Figure 3.19, is missing. Third, an event property is defined for the actor "ground station monitor" and the action "check" but the corresponding actual event definition is missing.

| The information shown below for aircraft is inadequate / incorrect. all aircraft has transponder |
| Missing actor for action generate in sentence s_005 |
| The property below is defined for 'ground station monitor' but there is no other reference to the corresponding action 'check' |
| → ground station monitor check for dangerous [ two or more aircraft in the same air space ] situation |
The warning messages produced from this evaluation are shown in Figure 4.9. These warnings include possible inconsistencies within the system's events, possible incomplete object information, and potentially unnecessary information.

<table>
<thead>
<tr>
<th>The action/object pair 'query - aircraft' has multiple actors defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>in sentence s_003 the actor of the pair is ground station monitor</td>
</tr>
<tr>
<td>in sentence s_009 the actor of the pair is controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The actor/action pair 'controller - query' has multiple objects defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>in sentence s_008 the object of the pair is ground station monitor</td>
</tr>
<tr>
<td>in sentence s_009 the object of the pair is aircraft</td>
</tr>
</tbody>
</table>

| No properties defined for: aircraft position |
| No properties defined for: controller |
| No properties defined for: database |
| No properties defined for: flight information |
| No properties defined for: graphic display |
| No properties defined for: transponder |

| No hierarchy relationships defined for: aircraft |
| No hierarchy relationships defined for: aircraft position |
| No hierarchy relationships defined for: controller |
| No hierarchy relationships defined for: database |
| No hierarchy relationships defined for: flight information |
| No hierarchy relationships defined for: graphic display |
| No hierarchy relationships defined for: ground station monitor |
| No hierarchy relationships defined for: transponder |

- aircraft has properties defined but participates in no operations

Figure 4.9: Sample Document Warnings

In addition to this error and warning report, a cross reference is provided for the information that is contained within the initial document. A sentence, object and action cross-reference is provided. A sample sentence cross-reference for the sentence "The monitor keeps a database that maintains this information" is shown in Figure 4.10. In addition, the cross-references for the object "aircraft" and the action "query" are also illustrated in Figure 4.10.
A regenerated requirements document is also produced by this evaluation process. This regenerated document presents the knowledge base information in an organized, object-based manner. All known information is detailed for each of the logical objects within the system, as illustrated in Figure 4.11 for the object "ground station monitor".

After examination of the document errors illustrated in Figure 4.8 and the regenerated document, the initial requirements document is revised to correct the known errors. The object "ground station monitor" is inserted as the actor of the sentence "A graphics display is generated from the current information". The event implied
ground station monitor
No hierarchy relationships exist between ground station monitor and any other objects within the system.

ground station monitor exhibits the following properties:
  ground station monitor check for dangerous [two or more aircraft in the same air space] situation

ground station monitor instigates the following events within the system:
  ground station monitor query aircraft for information
  ground station monitor keep database that maintain information
  ground station monitor update graphic display frequently [10 to 12 seconds]

ground station monitor is acted upon in the following events within the system:
  controller query ground station monitor for additional [data not normally displayed by the monitor] flight information

ground station monitor does not benefit from any actions within the system.

ground station monitor is visible to the following objects within the system:
  visible to 'controller' through the operation 'query'

ground station monitor is found in the original document in the sentences:
  s_002: the transponder transmits aircraft position to the ground station monitor
  s_003: the monitor can query an aircraft for information
  s_004: the monitor keeps a database that maintains this information
  s_006: the ground station monitor updates the graphics display frequently
  s_007: the monitor checks for dangerous situations
  s_008: the controllers may query the monitor for additional flight information

Figure 4.11: Regenerated Sample Document

by the event property in "The monitor checks for dangerous situations" is added, "The ground station monitor ... and checks this database frequently". Finally, the information for the object "aircraft" is updated. Two new properties are defined for this object in the sentence "All aircraft must have a radio and position indicator". The new requirements document is shown in Figure 4.12.
All aircraft must have a transponder. The transponder transmits aircraft position to the ground station monitor. The monitor can query an aircraft for flight information. The monitor keeps a database that maintains this information. The ground station monitor generates a graphics display from the current information. The ground station monitor updates the graphics display frequently and checks this database regularly. The monitor checks for dangerous situations. The controllers may query the monitor for additional flight information. Controllers may also query the aircraft for this information. All aircraft must have a radio and position indicator.

Figure 4.12: New Sample Requirements Document

4.4. Document Evaluation Summary

The evaluation process is responsible for analyzing the information contained within the initial requirements document. The generated knowledge base maintains the data in a format that permits this analysis. The system is now able to perform a formalized evaluation of this information, prior to the initiation of the actual specification process. By improving the quality of this document, the input into the specification process is also enhanced.

The evaluation process examines the information within the initial requirements document for reliability, testability and traceability. Reliability incorporates testing for completeness, consistency and necessity. It relies on both stand-alone analysis techniques and user-assisted analysis. Testability ensures that the information provided is presented in a structured, concise, self-descriptive and quantifiable form. Document traceability is also maintained by the system. Furthermore, any ambiguity within this data is removed during initial generation of the knowledge base.
This analysis generates two outputs for further examination by both the user and the specification team. First, a document report is generated. This report provides a listing of the document, a summary of the errors and warnings found within this document, and a complete cross-reference of the data. The second document produces a structured text containing information equivalent to the initial requirements document. This information defines all known features for the logical objects that compose the system, and is presented in a clear, organized manner.

4.5. The Generated Environment

An environment for the development of requirements specifications is created. This environment replaces the initial requirements document as the source for domain-specific information regarding the proposed system. The requirements document's information is formally transferred into the specification environment's knowledge base. The data now exists as a set of knowledge-based facts.

The specific functions and elements involved with the creation of this specification environment are shown in Figure 4.13. The initial English prose requirements document represents the input to the process. It is passed to the environment generation process. Here it enters the parse module and is converted into a set of parsed sentence structures. Access to an English dictionary is required by the parser during this processing. In addition, the parser relies heavily on user interaction to resolve any inherent ambiguities within the document. These parsed sentence structures are then passed to the conversion algorithm. Again, user interaction assists in the quantification and ambiguity resolution required during this phase. The generated
facts are used to construct a domain-specific knowledge base containing the information present within the initial requirements document.

Control is then passed to the document evaluation process. The evaluation routines analyze the information within the knowledge base. They provide both a stand-alone syntactic analysis of this information and an user-assisted semantic analysis. Two outputs are produced from this evaluation. The first is the complete document report and the second is the regenerated input requirements document. This knowledge base defines a formalized environment for the development of system specifications.

The development of this environment realizes three of the four objectives established for this research. Two of these objectives, enhancing information transferral and establishing a working base of knowledge for specification development, are realized during environment generation. The third objective, providing a method to improve the quality of the initial requirements document, is realized with the adoption of the formalized evaluation methodology. This methodology permits the evaluation and analysis of the initial document, and is therefore capable of improving the actual document submitted to the specification process.

Numerous benefits are realized from the adoption of this environment. First, formalized user involvement during the conversion of the initial requirements document is established. The user's domain-specific knowledge is utilized for the resolution of ambiguities that arise within this document, as well as during document evaluation. Second, a conversion algorithm is established to transfer the domain-specific knowledge into the specification process. This bridge between the user and the
specification team provides a formalized method for knowledge acquisition within the specification process. Third, a preliminary analysis of this information is now possible. Information that is incorrect, incomplete, inconsistent or unnecessary is identified before the process of system specification begins. Repetitive applications of this process define a cycle for the evaluation and improvement of these initial requirements. In addition, the notion of traceability is formally addressed. The knowledge base pro-

Figure 4.13 : Specification Environment
vides the necessary facilities to trace the user's information through the process of system specification development.

Finally, the establishment of a formalized environment for the development of specifications provides a proper input into the specification process. All other phases within the life-cycle are provided with a formalized, structured input. The design phase utilizes the formalized specification document as its input. The implementation phase utilizes the generated system design. The specification phase is the only phase within the life-cycle that does not establish a formalized input for its development. The knowledge-based environment, developed interactively with the user and the specification team, provides such an input.
Chapter Five

An Object-Oriented Specification Methodology

The fourth objective of this research is the creation of a paradigm for the development of object-oriented specifications. A technique that provides for the automatic generation of such specifications is illustrated. It assumes the existence of the knowledge-based specification environment that was developed to meet the first three research objectives. This knowledge-based environment defines a framework that permits the incorporation of object-oriented techniques into the requirements specifications phase.

The original object-oriented development scheme [Boo82] does not provide for the generation of system specifications. It relies on their generation through existing specification techniques. The heuristics involved in the identification of objects and their associated actions do not permit the establishment of a formalized object-oriented technique for requirements specification. This is primarily due to the inexact nature of the initial requirements document that is used as input to the specification process. The establishment of a formalized specification environment replaces this document with a structured knowledge base of information, providing a foundation for object-oriented specification development.

The original object-oriented development technique consists of a five-phase design methodology. These steps are identification of system objects, identification of system actions, definition of each object's visibility, establishment of all object
interfaces and implementation of each object. Three of these basic concepts, the identification of system objects, the identification of system actions, and the establishment of object visibility, are formally addressed during system specification. Conceptually, this analysis occurs at a higher level of abstraction than does its design counterpart. This specification methodology addresses these three tasks, within the context of the generated specification environment. The design-oriented tasks of establishing object interfaces and implementing these objects are not considered within the specification process.

5.1. The Object-Oriented Specification Technique

The technique developed in this research provides for the automatic generation of formalized system specifications. The generated specification contains all of the relevant information within the initial requirements document, structured into an object-oriented model of the system. The specification technique operates in an automatic manner. The development of an automatic technique for specification generation eliminates heuristic decision processes from the specification methodology. All information required for specification development is formally entered into the environment prior to the actual generation of the system specification. This also helps to ensure traceability of the specification model that is created through this process. A general overview to the specification technique is shown in Figure 5.1.

The input to this specification process is the knowledge-based environment developed in Chapter 3 and Chapter 4. It contains the user's domain-specific information regarding the proposed system, as presented in the initial requirements
document. It also contains additional information concerning the system’s logical objects and their visibility, added to the knowledge base during document evaluation.

The object-oriented specification technique uses a three-step methodology for specification development, as described in Figure 5.2. Identification of the objects that are included within the specification model occurs during the first phase of this process. A formalized definition for each of these objects is generated within the second phase. Finally, actual construction of the object-oriented specification occurs during the third phase in this methodology. These functions combine to provide an object-oriented specification model for the proposed system.
begin
execute Step One
for each object defined in Step One do
execute Step Two with
Object = object
Parent = nil
od
for each object defined in Step One do
execute Step Three
od
end

Step One : Identify the solution objects in the system
Step Two : Define the object models
Step Three : Generate the object models

Figure 5.2 : Top Level Specification Algorithm

5.1.1. Step One: Identification of the Solution Objects

Identification of the solution objects within the proposed system takes place within Step One. Solution objects represent the top-level logical objects within the system specification. These objects define the basic framework around which the system is constructed. The solution objects are derived from the set of logical objects that exist within the knowledge-based specification environment. To assist in this identification process, a classification scheme is developed for the system's logical objects. Using this classification scheme, a methodology for the identification of the system's solution objects is presented.
Object Classification

Each of the logical objects within the system are identified as either a potential live object or as a potential inert object. Live objects correspond to objects that participate in the system's events. Since all events defined for the system must be present in the system specification, the objects that participate in these events must also exist within the specification model. Not all live objects represent solution objects, however. The set of live objects defines a list of possible solution objects for the system.

Inert objects represent logical objects within the system that do not participate in any events. Inert objects are not included within the generated specification model. They can, however, mask the presence of a potential solution object, as discussed shortly. Their definition is therefore necessary to the process of identifying solution objects.

The live object and inert object sets retain the general type of their contained logical objects, and not the complete name of these objects. As a result, "object" is retained for the logical object "name of the object". This defines potential solution objects in their most generic sense. The derived objects within these generic objects, such as "name of the object", are elicited during definition and development of the more generic solution object.

Solution Object Identification

Three distinct tasks are required to generate the set of solution objects for the system, as described in Figure 5.3. The first task involves the identification of all live objects within the system. All logical objects that function as actors or recipients
within the system's events are identified as live objects. In addition, system entities that function as an event object are recognized as potential live objects. These event objects are included in the set of live objects if one or more of the following three conditions are satisfied. First, event objects that are visible to more than one actor within the system are included in the live object set. Such event objects can function externally to both of these actors, and thus require a separate entity to model their functioning. Second, event objects with properties that function as actors or recipients are added. Since these properties are each modeled as a separate entity within the system, an entity must exist within the system that contains these properties. Therefore, these event objects are added to the set of live objects. Third, properties or derived objects defined for an object that increase the event object's visibility also require the event object to be included in the set of live objects. The expanded visibility of this object now requires it to be modeled as a separate entity. Only event objects that are capable of being modeled completely within the confines of the associated event's actor are not included as live objects.

The second task in the identification of solution objects is the recognition of all inert objects within the system. Determination of inert objects is required for proper identification of the system's solution objects. Solution objects represent top-level objects within the system. Objects that are properties of other objects are not considered to be solution objects. Their modeling is generated recursively during construction of their top-level solution object. However, top-level objects that are inert, i.e. do not participate in the system's events, are not modeled as solution objects. Any live objects that are properties of these inert objects would not be modeled. There-
I) Determine potential solution objects, the live objects in the system
   for each object that participates in the system’s events do
      assert potential_live(object)
   od
   for each potential_live object do
      if the object is an
         event actor or
         event recipient or
         event object visible to two or more distinct actors or
         event object with properties that require separate modeling or
         event object with properties that increase its visibility then
            assert live(object)
      endif
   od

II) Determine the inert objects
   for each object that does not participate in the system’s events do
      assert potential_inert(object)
   od
   for each potential_inert object do
      if the object is not a property of a live object then
         assert inert(object)
      endif
   od

III) Identify the solution objects
   for each object in the set of live objects do
      if the object is a property of an inert object or
         the object is not a property of any live object then
         assert solution(object)
      endif
   od

Figure 5.3 : Step One of the Specification Process

fore, a recognition of the system’s inert objects permits the definition of their con­
tained properties as potential solution objects. Since these inert objects are deter­
mined during the identification of solution objects, the problems associated with the
hiding of potential solution objects within an inert object are eliminated.
To determine the system’s potential inert objects, all of the objects that are not involved within the system’s events are identified. Once this initial set is established, the true set of inert objects is generated by removing all objects that are properties of a live object within the system. Only the objects that could potentially provide a top-level covering of a true solution object are maintained as inert objects.

With the establishment of the live object and inert object sets, identification of the system’s true solution objects is performed during task three. The set of solution objects is generated from the set of live objects. All non-solution objects are eliminated from this set. This elimination process removes all live objects that are properties of other live objects, retaining only the highest-level live object as a potential solution object. As discussed previously, objects that are properties of inert objects are retained as potential solution objects. The remaining objects represent the set of solution objects for the proposed system.

5.1.2. Step Two: Definition of the Object Models

This specification technique is based on the concept of an object, specifically the solution objects defined for the system within Step One. Therefore, a complete definition of each object’s functionality within the proposed system is required. The definition of a solution object involves the resolution of seven distinct queries regarding the object. These are:

Where does the object fit in the organizational scheme?

What type of object is being defined?
What are the inherent properties of this object?

What events does this object initiate within the system?

What derived objects are defined within the context of this object?

What internal objects are present within this object?

What objects exist within the scope of this object?

Seven distinct tasks exist within the process of solution object definition, corresponding to the seven queries listed above. These tasks are shown in Figure 5.4. Each of the tasks considers a different aspect of the associated object’s functionality.

Figure 5.4: Solution Object Definition
The information obtained during these seven tasks is inserted into the specification environment's knowledge base as new facts. It is utilized during Step Three to construct the actual specification model. Six knowledge representation formats are defined for these facts, as illustrated in Figure 5.5. The *parent* fact identifies the parent object for the current object. The *type* fact defines the model used for the development of the associated object. *Property* and *procedure* facts detail the object's known information and functionality. *Derived* facts define more specific objects that exist within this generic entity. Finally, *internal* facts identify the objects that are contained within the current object.

<table>
<thead>
<tr>
<th>Object Information</th>
<th>Knowledge Base Fact Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent object</td>
<td>parent(Object, Parent_Object)</td>
</tr>
<tr>
<td>type</td>
<td>type(Object, Object_Type)</td>
</tr>
<tr>
<td>property</td>
<td>property(Object, Property_id, Property_Type, Location_Flag)</td>
</tr>
<tr>
<td>procedure</td>
<td>procedure(Object, Action, [Event List],</td>
</tr>
<tr>
<td></td>
<td>[Event_Object_and_Location_List],</td>
</tr>
<tr>
<td></td>
<td>[Constraint_List])</td>
</tr>
<tr>
<td>derived object</td>
<td>derived(Object, Derived_Object, Location_Flag)</td>
</tr>
<tr>
<td>internal object</td>
<td>internal(Object, Internal_Object, [Derived_Object_List])</td>
</tr>
</tbody>
</table>

Figure 5.5: Object Definition Fact Formats

Three additional types of facts are needed during object definition, as described in Figure 5.6. The *requires* fact defines the other solution objects in the system that are necessary for the current object's functioning. The *contains* fact identifies all objects that are subordinate to the current object in the specification model. The *local* fact defines objects that are modeled within the current object.
Object Information | Knowledge Base Fact Format
--- | ---
requires | requires(Object, Separate External Object)
contains | contains(Object, Contained External Object)
local | local(Object, Local Object)

*Figure 5.6: Additional Definitional Fact Formats*

**Parent Identification**

The object’s parent is determined during invocation of Step Two. The object and its parent are asserted into the knowledge base using the parent fact:

\[
parent \ (Object \ , \ Parent \ Object)
\]

The specification algorithm of *Figure 5.2* passes a *nil* parent during definition of the system’s solution objects. The last task within Step Two, Contained Object Definition, recursively calls Step Two for each object contained within the current object, passing the current object as the object’s parent.

**Type Definition**

Determination of object type constitutes the second task within Step Two. The object type defines a basic model for this object. The object-oriented development strategy assumes that all objects within the system are modeled as either *abstract state machines* or *abstract data types*. An abstract state machine defines a separate entity within the system that is considered capable of initiating independent actions. An abstract data type defines a specific data representation that is manipulated by the system. The system’s abstract data types represent passive objects. Such entities can not
Object type definition is determined by an analysis of the actions that an object initiates. Solution objects that independently initiate actions are modeled as abstract state machines. Solution objects that do not initiate any actions are modeled as abstract data types. This information is added to the knowledge base using the type fact, with object_type defined as either "adt" or "asm", in the type format of:

\[
\text{type ( Object, Object_Type )}
\]

The algorithm for determining object type is described in **Figure 5.7**.

![Figure 5.7: Determining the Object Type](image)

**Property Definition**

The third task in this process defines the object's properties. The property definition algorithm is illustrated in **Figure 5.8**. All of the physical properties for this object are identified and classified as either "simple" or "object". Simple properties, as described in Chapter 3, define inherent characteristics of the object. They include
adjectives and other non-object modifiers that are used to describe the object. Object properties define a property for the object in terms of the other objects that exist within the system.

During property definition, as well as procedure definition, the location of the defined entity must be established. The location of these entities is either "internal" to the current object or "external" to the object. Entities that can be modeled completely within the confines of the current object are defined as internal. Entities that contain characteristics that require separate modeling are defined as external to the current object.

All simple properties are considered internal to the object. Simple properties are asserted into the knowledge base with their corresponding Property_id, a Property_Type of simple, and Location_Flag set to internal, indicating they are defined within the current object:

\[ \text{property ( Object, Property_id, simple, internal )} \]

Object properties require a more extensive analysis. For all object properties, it is necessary to determine whether or not the property can be defined within the current object. External definition of the property is required if the property initiates actions or possesses properties of its own. Additionally, if the property is contained within another solution object, it is declared as external to the current object. This information is added to the knowledge base, with Location_Flag set to either internal or external:

\[ \text{property ( Object, Property_id, object, Location_Flag )} \]
Determine properties of the object $O$

- Find all physical properties that exist for the object
- For each property found do
  - Determine the Property Type of the property, simple or object
  - If the Property Type is simple then
    - Assert property($O$, Property_id, simple, internal)
  - Else (Property Type is object)
    - Determine if the property is "external"
      - If the object is a derived object for another object then
        - If the object is a solution object then
          - $Location\_Flag = \text{external}$
          - Assert requires($O$, object)
        - Else (the object is a non-solution object)
          - $Location\_Flag = \text{external}$
          - Assert contains($O$, object)
      - Else
        - If the object initiates actions or
          the object has properties of its own then
          - $Location\_Flag = \text{external}$
          - Assert contains($O$, object)
        - Else
          - $Location\_Flag = \text{internal}$
  - Endif
- Endif
- Assert property($O$, Property_id, object, $Location\_Flag$)

od

Figure 5.8 : Determining Object Properties

The location of all external properties, relative to the current object, is also determined at this time. This is represented through the use of the contains and requires facts. The contains fact defines all objects that are subordinate to the current object, corresponding to properties of the current object that initiate actions or have properties of their own. The requires fact defines all properties of the current object that are located outside the scope of the current object, that is, they exist as part of the other
solution objects in the system. These facts are asserted into the knowledge base, using the two auxiliary formats shown below:

\[
\text{contains (Object, Contained External Object)}
\]

\[
\text{requires (Object, Separate External Object)}
\]

**Procedure Definition**

The actions inherent to the current object are determined by the fourth task of Step Two, as illustrated in Figure 5.9. Unfortunately, it is not possible to base this definition on the events that contain the current object as their actor. A single logical event might be described in more than one location within the initial requirements document, resulting in multiple definitions for a single event. A grouping of the events that are initiated by the current object is required. This defines a set of "<event action / event object>" pairs. Each pair defined for the current object represents a unique procedure that is initiated by this object.

All relevant information regarding the object's procedures is accumulated by this task. First, the action listed within the "<event action / event object>" pair is assigned to the Action field. Next, the Event List is generated from all of the events within the knowledge base that correspond to this "<event action / event object>" pair. Third, for each of the objects involved in this procedure, its location relative to the current object is determined, as described in Figure 5.10. This location is either "external" to the current object or "internal", i.e. defined within the current object. External objects, visible to the current object through this procedure, include other solution objects, properties and derived objects contained within the other solution objects, as
Determine actions of the current object $O$

find all action/object pairs

that exist for this actor

for each pair found do

find all events

that match this action/object tuple

for each event found do

→ append this event to the Event_List

od

for each object involved in the action do

determine the location of the object(s) involved

→ see Figure 5.10

for each object and its location do

→ add this data to the Event_Object_and_Location_List

od

od

find all event properties

that exist for this action

for each event property found do

→ append it to the Constraint_List

od

assert procedure($O$, $Action$, $Event_List$, $Event_Object_and_Location_List$, Constraint_List)

od

Figure 5.9: Determining Object Procedures

well as separate properties and derived objects defined for the current object. Internal objects correspond to both internal properties of the current object and logical objects that are only acted upon by the current object. The objects and their locations, either internal or external, are accumulated into the Event_Object_and_Location_List. Finally, any constraints on the procedure are located. These constraints are obtained from the event properties defined within the knowledge base. The constraint generation process matches the current object with the event property’s object and the current action with the event property’s action. All constraints for a given event are
accumulated in the *Constraint_List*. After accumulating this information, a procedure fact is inserted into the knowledge base using the following format:

```
procedure ( Object, Action, Event_List, 
            Event_Object_and_Location_List, Constraint_List )
```

The assertion of the fact that the current object *requires* any external event objects referenced in the event is also generated. In addition, the fact that internal event objects that are not properties of the current object are local to the current object is inserted into the knowledge base, using the *local* fact.

**Derived Object Definition**

All derived objects within the current object are defined in the fifth process of Step Two, as illustrated in Figure 5.11. Derived objects define a specific instantiation of the more generic current object. These derived objects provide a method for decomposition of the generic object into specific entities. The specific object contained within the generic object is stored in *Derived_Object*. Derived objects are either completely contained within the current object or defined externally, depending upon whether or not they initiate actions or have properties of their own. Derived objects that are contained within the current object have *Location_Flag* set to *internal*, while derived objects that require separate modeling set *Location_Flag* to *external*. Information regarding the derived objects is accumulated and inserted into the knowledge base using the derived fact:

```
derived ( Object, Derived_Object, Location_Flag )
```
Determine the Location of the object(s) involved

if the object a solution object then
  \( \text{Location} = \text{external} \)
  \( \rightarrow \) assert requires\((O, \text{object})\)
elseif the object is a property then
  if it is a property of object \( O \) then
    if it is a "external" property then
      \( \rightarrow \text{Location} = \text{external} \)
    else
      \( \rightarrow \text{Location} = \text{internal} \)
    endif
  endif
else { the object is a property of another object }
  \( \rightarrow \text{Location} = \text{external} \)
  \( \rightarrow \) assert requires\((O, \text{object})\)
endif
elseif the object is a derived object for another object then
  if the other object is a solution object then
    \( \rightarrow \text{Location} = \text{external} \)
    \( \rightarrow \) assert requires\((O, \text{object})\)
  else
    \( \rightarrow \text{Location} = \text{internal} \)
    \( \rightarrow \) assert local\((O, \text{object})\)
  endif
else
  \( \rightarrow \text{Location} = \text{internal} \)
  \( \rightarrow \) assert local\((O, \text{object})\)
endif

Figure 5.10: Determining Object Location during Procedure Definition

Additionally, the fact that the current object contains all of its external derived objects is asserted into the knowledge base using the contains fact.

Internal Object Definition

Objects internal to the current object are realized during task six within the object definition step. Such objects are referenced by the procedures that the current object initiates. They are not internal properties of the object, as internal properties of
Determine all derived objects for the current object $O$

find all derived objects

that exist for the current object

for each derived object found do

if this object has properties of its own or it initiates events on its own then

→ assert contains($O$, object)

→ Location_Flag = external

else

→ Location_Flag = internal

endif

→ assert derived($O$, object, Location_Flag)

od

Figure 5.11: Determining Derived Objects

the current object are defined during generation of property facts. However, they are only acted upon by the current object. They are therefore capable of being completely modeled within the confines of the current object. Internal objects are accumulated through a search of the knowledge base, identifying all local facts defined for the current object that were generated during processing of the current object's procedures. All derived objects for this internal object are also located and accumulated in the internal object's Derived_Object_List. This information is added into the knowledge base using the internal fact:

internal (Object, Internal_Object, [Derived_Object_List])

The algorithm for determining internal objects is illustrated in Figure 5.12.

Contained Object Definition

The generation of contained objects is the final task within Step Two. This process is illustrated in Figure 5.13. All objects that the current object contains are
Determine all internal objects of the object \( O \)
find all the assertions \( local(O, \text{object}) \)
that exist in the knowledge base
for each item in this list do
find all derived objects
that exist for this local object
\( \rightarrow \) assert \( internal(O, \text{object}, [\text{derived objects}]) \)
enddo

Figure 5.12: Determining Internal Objects

retrieved from the knowledge base. These objects are recursively passed to Step Two
for object definition. The current object is also passed, representing the parent of this
object. Contained objects include both external properties of the current object that
initiate events or have properties of their own and also external derived objects of the
current object. As this process is recursive in nature, all nested objects within the ini-
tial set of solution objects are generated.

Build all "contained" properties/types for current object \( O \)
find all the assertions \( contains(O, \text{object}) \)
that exist in the knowledge base
for each object found do
call Step Two with:
Object = \( \text{object} \)
Parent = \( O \)
enddo

Figure 5.13: Definition of Contained Objects

In addition to these seven tasks, Step Two also establishes the visibility of the
solution objects within the system model. This is performed throughout the
definitional processes, by the use of \( requires \) assertions into the knowledge base.
Visibility of an object is defined as the set of objects within the system model that
must be accessible to the given object. This visibility provides a mechanism for determining object interactions and dependencies. Only the interactions defined within this specification model require consideration during the remainder of the software development life-cycle.

At the end of Step Two, the knowledge base contains all necessary information for the generation of an object-oriented specification. The objects and actions present within the system are now formally defined. Their type, properties, functionality, visibility and contained features are also described within the knowledge-based specification environment.

5.1.3. Step Three: Generation of the Object Models

Actual construction of the specification occurs during the third step of specification generation. The information necessary to construct this specification is now present within the specification environment's knowledge base. The generation process must accumulate, organize and output this information in a coherent manner, presenting it in an object-oriented format. The structure of the object modules generated is illustrated in Figure 5.14.

The developed specification is presented in a modular form. Each module represents the solution-space model for one of the solution objects identified within Step One. Each provides a detailed description of the functionality of the associated solution object. In addition, the visibility of this object is established. The structure of the modules for objects contained within the solution objects is identical to the format illustrated in Figure 5.14, with the exception that the module header identifies the
package  OBJECT is
  model   Abstract Data Type or Abstract State Machine
  requires All other visible packages
  contains All "package-parts" contained within this OBJECT
  property PROPERTY:type [ is separate ]
  procedure ACTION [ Obj1:location ]+
                   [ constraint event property ]*
  derived   DERIVED_OBJECT [ is separate ]
  internal  INTERNAL_OBJECT [ with derived_objects ]*
end   OBJECT;

Figure 5.14 : Object Module Format

parent object for this contained object, as shown below:

    package-part OBJECT belongs to PARENT

The technique employed during specification generation, shown in Figure 5.15, is applied to each of the system’s solution objects. The object’s parent, type, properties and procedures, derived objects and internal objects are obtained from the knowledge base. The "is separate" clause on properties and derived objects indicates objects that are defined as "external" to the current object. Additional information accompanies the property and procedure definitions for a given object. Specifically, the text from the initial requirements document that defines these properties or events is produced as a comment entry, along with the formalized definition. All objects
contained within this object and all other objects required by this object’s functionality are also identified. Objects contained within this object have their own specification module generated in a recursive manner. This information is then structured and presented in the format shown in Figure 5.14.

<table>
<thead>
<tr>
<th>Accumulate object information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>find</strong> all the relevant assertions</td>
</tr>
<tr>
<td>that apply to the current object O</td>
</tr>
<tr>
<td>parent(O, Parent)</td>
</tr>
<tr>
<td>type(O, Object_Type)</td>
</tr>
<tr>
<td>property(O, Property_id, Property_Type, Location_Flag)</td>
</tr>
<tr>
<td>procedure(O, Act, [Event List], [List of Event_Object_and_Location List], [Constraint List])</td>
</tr>
<tr>
<td>derived(O, Derived_Object, Location_Flag)</td>
</tr>
<tr>
<td>internal(O, Internal_Object, [Derived_Object_List])</td>
</tr>
<tr>
<td>requires(O, External_Object)</td>
</tr>
<tr>
<td>contains(O, Contained_Object)</td>
</tr>
</tbody>
</table>

Format the object information as illustrated in Figure 5.14

Print the object model

Generate models for the objects contained within this object

find all the assertions contains(O, object) that exist for the current object O for each object in the contains list do generate that object model (recursively call this process) od

Figure 5.15 : Step Three of the Specification Process

After execution of the object-oriented specification methodology, a functional model exists for the proposed system. This model is based on the information contained within the initial prose requirements document. It defines the basic objects that
comprise the system, and also illustrates their proposed functionality. In addition, the visibility of these objects is defined. This model is intended for use as input to a formalized design methodology. The system model can be utilized with any number of possible design techniques. It is recommended, however, that an object-oriented methodology [Boo86, Boo87] is adopted for system design.

5.2. Generation of an Object-Oriented Specification for the Sample Document

An object-oriented specification is generated for the sample requirements document as it exists in Figure 4.12. The paradigm uses the information present within the environment's knowledge base to generate this specification.

Solution Object Identification

The first step in the process is the identification of solution objects for this system. The logical objects within the system are defined as either potential live objects or potential inert objects. Figure 5.16 indicates the classification of each of the ten logical objects within the system. Eight objects participate within the system's events, and are identified as potential live objects. These are "aircraft", "aircraft position", "controller", "database", "flight information", "graphics display", "ground station monitor" and "transponder". From this list, five live objects are identified, "aircraft", "controller", "database", "ground station monitor" and "transponder". The event object "aircraft" is visible to multiple actors, and it contains a property, "transponder", that initiates actions on its own. The event actors "database", "controller" and "ground station monitor" are also live objects. The three objects "aircraft
"position"", "flight information" and "graphics display" are all event objects that are only visible to one event actor, and are not considered live objects. Two potential inert objects are defined, "position indicator" and "radio". Neither of these two objects participate in the system's events. However, both are properties of a live object and are thus not defined as inert objects. Four solution objects are defined for the system. These are the four live objects that are not properties of any other object, "aircraft", "controller", "database" and "ground station monitor". The object "transponder" is a property of another live object, and therefore not a solution object.

<table>
<thead>
<tr>
<th>Object</th>
<th>Potential Live Object</th>
<th>Actual Live Object</th>
<th>Potential Inert Object</th>
<th>Actual Inert Object</th>
<th>Solution Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>aircraft</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>aircraft position</td>
<td>yes</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>controller</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>database</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flight information</td>
<td>yes</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>graphics display</td>
<td>yes</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ground station monitor</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>position indicator</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>radio</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>transponder</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.16 : Logical Objects within the Sample Document**

**Solution Object Definition**

Once the solution objects are defined for the system, all relevant information regarding these objects is defined within the knowledge base. **Figure 5.17** illustrates the information that is accumulated for the solution object o_001, "aircraft". The type fact defines "aircraft" as an abstract data type. The object o_002,"transponder",
is contained within this object. Three object properties are defined for this object. The first property, p_001, is an external property, "transponder", that initiates actions of its own. The last two properties, p_003, "position indicator", and p_004, "radio", are internal to this object. To aid in comprehension, the sentences that define these properties are also listed in the table.

| type( [o_001], adt ). |
| contains( [o_001], [o_002] ). |
| property( [o_001], p_001, object, external ). |
| → All aircraft have a transponder. (p_001) |
| property( [o_001], p_003, object, internal ). |
| → All aircraft must have ... and position indicator. (p_003) |
| property( [o_001], p_004, object, internal ). |
| → All aircraft must have a radio... (p_004) |

**Figure 5.17 : Definitional FactsGenerated for "aircraft"**

The information detailed in Figure 5.18 is generated for the object o_004, "ground station monitor". As it initiates actions within the system, it is modeled as an abstract state machine. Two external objects are required for its functioning, o_001 - "aircraft" and o_006 - "database". Five distinct procedures exist for this object. The action, a list of event ids that correspond to this function, a list of objects involved in the function and their locations, and a list of the constraints placed on these functions are defined for each procedure. The sentences that define these procedures are also listed in the table. The last procedure also has an event property defined for the event, property p_002 - "the monitor checks for dangerous information". Finally, the
definition of the internal object o_007, "graphic display" is generated.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>type( [o_004] , asm ).</td>
<td>-4 The monitor (o_004) can query (a_002) aircraft (o_001) for flight information.</td>
</tr>
<tr>
<td>requires( [o_004] , [o_001] ).</td>
<td></td>
</tr>
<tr>
<td>requires( [o_004] , [o_006] ).</td>
<td></td>
</tr>
<tr>
<td>internal( [o_004] , [o_007] , [] ).</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.18: Definitional Facts Generated for "ground station monitor"

Solution Object Model Generation

Packages are generated for each of the four solution objects, "aircraft", "controller", "database" and "ground station monitor". In addition, the model for the contained object "transponder" is generated during creation of the parent model "aircraft". The complete specification model is as follows:
package aircraft

model Abstract Data Type
requires <>
contains transponder

property transponder:object is separate
   -- all aircraft has transponder

property position indicator:object
   -- all aircraft has position indicator

property radio:object
   -- all aircraft has radio
end aircraft

package-part transponder belongs to aircraft

model Abstract State Machine
requires <>
contains <>

procedure transmit [ aircraft position:internal ]
   -- transponder transmit aircraft position to ground station monitor

internal aircraft position
end transponder

package controller

model Abstract State Machine
requires aircraft, ground station monitor
contains <>

procedure query [ ground station monitor:external ]
   -- controller query ground station monitor for additional [ data not normally displayed by the monitor ] flight information

procedure query [ aircraft:external ]
   -- controller query aircraft also for flight information
end controller
package database
model Abstract State Machine
requires <>
contains <>
procedure maintain [ flight information:internal ]
-- database maintain flight information
internal flight information
end database

package ground station monitor
model Abstract State Machine
requires database, aircraft
contains <>
procedure query [ aircraft:external ]
-- ground station monitor query aircraft for flight information
procedure keep [ database:external ]
-- ground station monitor keep database that maintain flight information
procedure generate [ graphic display:internal ]
-- ground station monitor generate graphic display from current [ the latest data received ] flight information
procedure update [ graphic display:internal ]
-- ground station monitor update graphic display frequently [ 10 to 12 seconds ]
procedure check [ database:external ]
constraint ground station monitor check for dangerous [ two or more aircraft in the same air space ] situation
-- ground station monitor check database regularly [ every 2 seconds ]
internal graphic display
end ground station monitor
5.3. Summary

An object-oriented paradigm is developed for use within the process of requirements specifications. This methodology utilizes the knowledge-based specification environment that is constructed from the initial requirements document. The information within the environment's knowledge base provides a reliable, unambiguous framework for the development of these specifications.

The methodology provides a three-step process for the development of object-oriented specifications. First, the solution objects within the system model are identified. Second, all relative information regarding these solution objects is identified and inserted into the specification environment's knowledge base. This information includes data regarding the object's parent, type, properties, procedures, derived objects and internal objects. Third, the information accumulated during the second step is used to generate an object-oriented specification model of the system.

This paradigm realizes a number of benefits to the process of specification development. First, the specification technique utilizes an automatic algorithm for actual specification generation. This eliminates the use of heuristics during the specification process. The user and specification team interact with this process only during generation of the environment's knowledge base. Once this knowledge base is established, the specification process relies completely upon the information contained within it. No informal decision processes are permitted.

Second, the technique presents an object-oriented approach to the problem of specification development. The informal nature of the initial requirements document hinders the use of object-oriented approaches by conventional specification
techniques. By providing a formalized environment for specification development, it is possible to incorporate an object-oriented methodology into this phase of the software development life-cycle.

Third, the specification provided is an operational model of the system. The initial requirements document is expressed in an operational fashion. The process of system design also utilizes an operational approach to system development. By providing an operational specification, the translation required between these stages of development is minimal. While the developed specification is an operational model of the system, it still contains the functionality of the proposed system. No design decisions are included within this document.

Finally, traceability within the software development life-cycle is enhanced. The specification environment captures the information contained within the user's initial requirements document. The knowledge base permits the tracing of this data as it proceeds through the specification development process. It also permits the information within a defined specification model to be traced back to its origin within the requirements document.
Chapter Six

Summary

This research defines a knowledge-based environment for the creation of object-oriented system specifications, as shown in Figure 6.1. The overall specification paradigm consists of three processes. First, a method for the generation of the specification environment is provided. Second, a technique for the evaluation of the information contained within the environment’s knowledge base is illustrated. Third, an algorithm that uses this knowledge-based environment for the automatic generation of object-oriented specifications is presented. These three processes combine to form a paradigm for the development of object-oriented specifications.

6.1. The Specification Paradigm

The generation process takes as its input an initial English prose requirements document. This document contains the desired functionality of the proposed system, as defined by the user. The document is passed to a parse routine that converts its natural language text into a sequence of parsed sentence structures. The parser requires both a dictionary of the English language for word identification and the ability to interact with the user for the resolution of grammatical ambiguities. The parsed sentence structures are then passed to a conversion algorithm, where they are converted into an equivalent set of knowledge base facts. The conversion algorithm utilizes information contained within the parsed sentence structures to guide this process.
Figure 6.1: The Object-Oriented Specification Paradigm

It also defines a standardized representation format for the storage of this data. The process relies on user interaction to quantify any abstract terminology used within the document, to assist in the semantic interpretation of conjunctions and disjunctions within the parsed sentences, and to resolve inexact object references within the document. The knowledge base that is generated during this process contains a set of facts
representing the domain-specific information present within the initial requirements
document.

Once the information in the initial requirements document is converted into a set
of knowledge-based facts, a formalized evaluation of this information is possible.
This analysis process examines the contained data for reliability, testability and tra­
ceability. The notion of reliability includes examining the overall completeness of the
information present within the document, evaluating the consistency of this data, and
determining the necessity of the information provided. This process relies on user
interaction for assistance during the evaluation of information consistency and
correctness. Two outputs are produced during this evaluation procedure. First, a
document report is created for this requirements document. The report contains a list­
ing of the original document, its errors, warnings and a complete cross-reference.
Second, a structured, coherent, self-descriptive requirements document is regenerated
from this data.

Using this knowledge-based specification environment, a formalized
specification process is defined. The specification developed through this process pro­
vides an object-oriented model of the system. The specification process is a com­
pletely automated procedure, relying only on the information contained within the
associated knowledge base during construction of the system specification.

6.2. Limitations of the Specification Paradigm

Various limitations are realized during the implementation of this specification
methodology. First, restrictions are placed upon the input grammar to the parse
module. The text accepted by the parser represents a subset of the entire English language. Pronouns, certain types of sentence transformations, and non-standard noun phrases are all considered outside the legal input domain. Second, the parser functions in a purely syntactic manner. The resulting parsed sentence structures capture the syntactic content of their original sentences, yet lack any semantic interpretation of this information. Third, the analysis routines provided by the system are primarily syntactic in their nature. A semantic evaluation of this information must be performed by the user.

None of the limitations realized within the system represent inherent inadequacies within the general methodology. The limitations noted are a result of current implementation decisions, and are not considered inherent obstacles blocking future development. The emphasis of this research is the development of an object-oriented specification methodology. While the generation of a working specification environment comprises the foundation of this paradigm, the major concern is the development of the actual object-oriented specification methodology. A complete development of the associated specification environment is not vital to this research. The feasibility of such an environment, however, must be demonstrated. As a result, the major emphasis during the processes of environment generation and evaluation is a demonstration of the ability to establish this environment, and not the development of an exhaustive solution to this problem.

As an example, consider the limitations placed upon the system by the parse module. Natural language parsing techniques are capable of providing a syntactic parse for the complete English language [Tic87]. The current system parser
encompasses a subset of the complete English grammar. Expanding this parser does not expand the scope of the research. The benefits realized through the adoption of a formalized parse mechanism are illustrated within the current parse module. However, this does not imply that expansions to the parser would not enhance the system. Extensions to the parser provide a "friendlier" environment for the generation of the initial requirements document. Less concern is placed on document style, and more on document content.

Therefore, the limitations placed upon the system also illustrate the overall strength of the underlying methodology. The restrictions noted within the current methodology are a result of the nature of the current implementation. Future extensions to this work can reduce or eliminate the scope of these limitations.

6.3. Benefits of the Specification Paradigm

Each of the individual phases defined within the overall methodology realizes inherent benefits to the specification process. Separately, each provides improvements to current specification processes. Combined, they form a knowledge-based paradigm for the generation of object-oriented system specifications.

The construction of the knowledge-based specification environment realizes two major benefits. First, the user interface into the specification process, i.e. the bridge between the user and the specification team, is now formally defined. By providing a formalized user interface, the role of the user within the specification process is enhanced. The user serves as an aid to the process of resolving grammatical ambiguity within the document, assists in the task of quantifying the document's abstract
terminology, and acts as a semantic interpreter during the conversion of these sentences into knowledge base facts. Second, the use of a standardized knowledge base of information provides a formalized representation for the domain-specific information contained within the requirements document. This representation scheme presents the data in a format that is void of any grammatical ambiguity. In addition, this information is accessible to the specification team. The knowledge base defines a structured mechanism for the access, modification and addition of information regarding the proposed system.

The environment evaluation methodology provides a formalized analysis technique for the information contained within the initial requirements document. Previously, such an automated evaluation was not possible, as the informal nature of the initial requirements document prohibited this analysis. An evaluation of the information contained within the specification environment's knowledge base enhances the early detection of inadequate or incorrect information within the requirements document. Incomplete, inconsistent and unnecessary information is identified and eliminated from the document before a formalized specification process is initiated. As a result, the overall quality of the input to the actual specification process is improved.

The actual specification process generates a system model from the environment's knowledge base. A number of benefits are realized through this specification paradigm. First, an object-oriented approach to the task of system specification is permitted. Such an approach provides a minimal transition between the real-world model and solution-space model of the system. An object-oriented technique also provides benefits that are realized during the modification and
maintenance of the developed system. Second, the knowledge-based environment provides an operational framework for the generation of system specifications. The operational nature of the environment mirrors the initial requirements document. It provides benefits during the verification of the developed specification, as direct execution and evaluation of this specification is possible. Furthermore, the specification is generated in a completely automated manner. No informal or heuristic decision techniques enter into the specification process, as it relies completely upon the information that is contained within the generated knowledge base. All data used during the specification process must be formally entered into the environment's knowledge base prior to the specification process.

Finally, the concept of traceability within the specification process is enhanced with the development of this specification environment. The ability to trace domain-specific information through the process of system specification assists during verification of the developed specification. It provides capabilities for the explanation and justification of any module within the system. In addition, maintenance of the developed system is enhanced. Modifications to a particular set of requirements are mapped directly into the appropriate specification modules. This allows the maintenance team to identify the exact location for all changes necessary to the existing system.
The object-oriented specification paradigm presented in this work defines a formalized methodology for the development of system specifications. The paradigm creates a knowledge-based specification environment, and then defines an object-oriented specification technique for use within this environment. With the establishment of this methodology as a viable technique for the development of system specifications, a number of potential enhancements and future research directions are illuminated.

First, extensions of the knowledge-based environment to include general background information regarding the application domain should be considered. The knowledge base established by the environment contains no domain-specific information regarding the proposed system other than what is supplied within the initial requirements document. While the system presents a working method for specification development, it is incapable of providing an extensive semantic analysis regarding the overall consistency or correctness of the initial requirements document’s information.

A technique that incorporates the domain-specific information of an expert in the application area would provide additional benefits. By incorporating the background knowledge of an expert, the system could provide an automated analysis of this docu-
ment regarding both its consistency and overall correctness. This background knowledge includes both general facts regarding the application domain as well as rules governing the behavior and actions of the well-known entities within the domain.

An increase in the background information present within the knowledge base would improve the overall document evaluation process. Such an analysis is currently hindered by the need for a semantic interpretation of this data. In addition to the improved evaluation capabilities regarding document consistency, completeness and necessity, new checks concerning correctness and other relevant characteristics of the document could be incorporated into the evaluation process. As the evaluation methodology is strengthened, the overall quality of the requirements document is improved. Since this document represents the input to the object-oriented specification paradigm, it would therefore enhance the overall reliability of the generated specification model.

A second enhancement to the existing methodology is a continuation of the generated knowledge base throughout the remainder of the software development lifecycle. Knowledge-based techniques, such as KBEmacs and IDeA, have proven to be an effective tool for assisting in the process of system design. The scope of the knowledge-based environment would first be expanded to include the design phase. This would provide the design team with an intelligent assistant that contains both the proposed system specification and its origin. The design generated from this specification model would also be stored within the knowledge base.
This continuation of the knowledge-based environment would culminate in the maintenance phase of the life-cycle. At this point, information regarding the total software development process would be contained within the generated knowledge base. The knowledge base would provide an optimum guidance tool for any necessary maintenance or modifications to the system. During modification, it would be capable of identifying exactly what modules require updating, as well as identifying any side-effects introduced by this change. The maintenance of the developed system would no longer resemble an iterative sequence of patches that fix the original problem, the new problems introduced with the corrective patch, problems discovered with the patches' patches, and so on. Instead, the code requiring change could be located and correctly updated with minimal iteration.

A third possible enhancement to the methodology is the use of a more formalized specification language during actual specification development. As described in Chapter 2, formal methodologies for specification development permit a rigorous verification and validation of the developed specification. However, such methodologies are lacking in their ability to organize or guide the overall specification development process.

The environment created in this methodology generates a knowledge base containing the initial requirements document's information. Instead of using this knowledge base as the input into an object-oriented specification technique, it would be used to generate a formal system specification. The automatic nature of the specification algorithm would be maintained, thus providing the guidance mechanisms
lacking in standard formal methodologies. The structured nature of the knowledge-based environment established by this paradigm provides the organizational capabilities missing in a formal specification methodology.

The underlying logic of a formal specification technique, such as the Larch methodology, is easily captured within a PROLOG-based knowledge base. The inherent features of the domain-specific information, i.e. its object-oriented nature and its operational characteristics, would still be maintained. However, modifications would be necessary to the process of knowledge base generation. The conversion algorithm that translates the parsed sentence structures into a set of facts would be modified to reflect the formal specification methodology's notation. Due to the structured nature of the formal specification language, the algorithm would be forced to realize a more rigorous technique during its conversion process. It would be necessary to reject sentences that do not meet established guidelines.

Another potential research path would extend the scope of the system into the process of generating the actual requirements documents. As the requirements document represents the input to this specification methodology, improvements to this document would permit a more rigorous examination of its information during document evaluation. These improvements can either occur after the document generation process is complete, as in the current methodology, or during the actual generation process.

A document generator would be defined that acts as an intelligent assistant to the process of document creation. This generator would function as a knowledge acquisi-
tion tool. Methods for the extraction of all relevant information possessed by the user would be incorporated into the document generator. An evaluation of the information in the document would be performed as it is entered, prompting the user for additional or clarifying data as required. Such a document generator would require an extensive examination regarding the semantics of the user’s input. The generator must possess the capabilities to comprehend the semantic meaning of the document’s information. Only after the sentence’s meaning is determined would the system be able to determine any missing information that is relevant to the given sentence.

Finally, a related research area has emerged as a result of this work. This is an attempt to identify and capture the decision processes that are applied during the process of system specification. The developed specification contains essentially the same information as does the initial requirements document produced by the user. However, the two are not equivalent in terms of their ability to guide a design process. Therefore, the structure and presentation format inherent within a formalized specification should provide assistance to the process of system design. A recognition of the inherent features that provide this assistance would reveal new insights for the development of an optimal specification methodology.

This research path is part of a larger overall concern within the software engineering discipline. The software development life-cycle starts with a real-world model of the system as its input and produces a software model of this system as its output. A multitude of distinct transformation sequences exist that map this input into this output. Each phase of the life-cycle contains its own set of unique transformation
sequences that map the input to that phase into the corresponding output. Nevertheless, for a given input, the outputs obtained from various transformation sequences should all be functionally equivalent. Therefore, the question of what constitutes an optimal transformation sequence arises. Can one sequence of transformations be shown to be ideal for a given class or classes of input? If so, what characteristics are present within this input class that promote this sequence of transformations? A study into the problem of determining the relative effectiveness of the various transformation methodologies could provide insight into some of the major problems inherent within the development of large-scale software systems.
Bibliography


Specifications using CLEAR, in *The Correctness Problem in Computer

[Buz85] Buzzard, G. D. and Mudge, T. N., Object-Based Computing and the ADA

[Cha86] Charette, R. N., in *Software Engineering Environments*, McGraw-Hill,
1986.

[Che76] Chen, P. P. S., The Entity-Relationship Model: Toward a Unified View of
Data, *ACM Transactions on Database Systems* 1, 1 (March 1976), 9-36.

and Control* 2 (1959), 137-167.

[Cox84] Cox, B. J., Message/Object Programming: An Evolutionary Change in

[Dah66] Dahl, O. J. and Nygaard, K., SIMULA - an algol-based simulation


[Fin83] Finin, T. W., The Planes Interpreter and Compiler for Augmented
Transition Network Grammars, in *The Design of Interpreters, Compilers,
and Editors for Augmented Transition Grammars*, Springer-Verlag, 1983.


Heidorn, G. E., Automatic Programming Through Natural Language


Engineering, 1985, 83-89.


Appendix

A Sample Execution of the Specification Paradigm

The specification paradigm is applied to the following input requirements document. The interactive parse session is illustrated for this document, followed by a listing of the knowledge base facts that the document generates. The document report and regenerated requirements document are also displayed. Finally, the generated specification model for this document is presented.

The Widget Warehouse Company sells products to its customers. The company has a clerk who handles the customers and their orders. Customers order these products by phone. Each order has a customer name, customer address, product id, quantity and delivery date. The clerk allocates stock and ships the orders. The clerk bills the customer with the shipment. The clerk maintains the current inventory and purchases out-of-stock products. Each product has a product id, inventory price and customer price. The customer can modify the quantity and delivery date. The customer may cancel their current order.

Document Parsing Session

Parsing of the initial requirements document generates the following dialogue session. All system responses are shown in italic font, while the information provided by the user is shown in bold font.
Parsing the sentence (the widget warehouse company sells products to its customers)

What does the prepositional phrase (to its customers) modify?
   a. The noun products
   b. The predicate itself
   c. None of the above
   Select 'a', 'b' or 'c': b

Parsing the sentence (the company has a clerk who handles the customers and their orders)

Is the use of (company) an inexact reference to the object (widget warehouse company)?
   Please select one of 'y', 'n': y

Do the noun phrases in (the customers and their orders) function
   (a) separately
   (b) together
   Please select one of 'a', 'b': a

Parsing the sentence (customers order these products by phone)

What does the prepositional phrase (by phone) modify?
   a. The noun products
   b. The predicate itself
   c. None of the above
   Select 'a', 'b' or 'c': b

Parsing the sentence (each order has a customer name xxx customer address xxx product id xxx quantity and delivery date)

Do the noun phrases in (a customer name and customer address and product id and quantity and delivery date) function
   (a) separately
   (b) together
   Please select one of 'a', 'b': a
Parsing the sentence (the clerk allocates stock and ships the orders)

Parsing the sentence (the clerk bills the customer with the shipment)

ATN Error: Undefined word: bills
Enter word definition, 'help' or 'exit': (bill verb -s-ed-ing features (trans))

What does the prepositional phrase (with the shipment) modify?
   a. The noun customer
   b. The predicate itself
   c. None of the above
   Select 'a', 'b' or 'c': b

Parsing the sentence (the clerk maintains the current inventory and purchases out-of-stock products)

The modifier 'current' is inherently ambiguous
   No definition currently exists for current

Building new definition for the modifier 'current'

   The definition may be in one of the two following two forms
   a) units of measure / upper bound / lower bound
   b) an english description of the modifier

   For english description, specify the units of measure as 'english'

   In what units is the modifier measured? english
   Give an precise english description encased in a set of parenthesis
   -> (the amount in stock at the start of the day)

Is the following information correct for 'current'?
   -> (the amount in stock at the start of the day)
   Please select one of 'y' 'n': y
Parsing the sentence (each product has a product id xxx inventory price and customer price)

Do the noun phrases in (a product id and inventory price and customer price) function
(a) separately
(b) together
Please select one of 'a' 'b' : a

Parsing the sentence (the customer can modify the quantity and delivery date)

Do the noun phrases in (the quantity and delivery date) function
(a) separately
(b) together
Please select one of 'a' 'b' : a

Parsing the sentence (the customer may cancel their current order)

The modifier 'current' is inherently ambiguous
The following definitions exist for current
0. The definitions below are not correct, build a new definition
1. -> (the amount in stock at the start of the day)
Please select one of '0' '1' : 0

Building new definition for the modifier 'current'

The definition may be in one of the two following two forms
a) units of measure / upper bound / lower bound
b) an english description of the modifier

For english description, specify the units of measure as 'english'

In what units is the modifier measured ? english
Give an precise english description encased in a set of parenthesis
-> (any outstanding unfilled order)

Is the following information correct for 'current' ?
-> (any outstanding unfilled order)
Please select one of 'y' 'n' : y
Parsing completed.

Completing generation of the knowledge base facts ... done.

Loading prolog for next phase

C-Prolog version 1.5
[ Restoring file system.data ]

Parsing the input requirements document is now complete. The input document has been converted into a set of knowledge base facts. You may now either proceed to check this knowledge base for reliability, testability, and traceability OR exit the program to make corrections to the initial requirements document.

To continue with the checking, type 'go.'
To exit the program type 'exit.'

? go.

Knowledge base checking begins ...

Interactive checking of object properties

---------------------------------------------------------------------

The following properties are defined for order

    order has delivery date
    order has quantity
    order has product id
    order has customer address
    order has customer name

Is the information shown here correct?
please reply with either 'y.' or 'n.' -> y.

---------------------------------------------------------------------
The following properties are defined for product

- product has customer price
- product has inventory price
- product has product id

Is the information shown here correct?
please reply with either 'y.' or 'n.' -> y.

The following properties are defined for widget warehouse company

- widget warehouse company has clerk who handle customer

Is the information shown here correct?
please reply with either 'y.' or 'n.' -> y.

End of interactive checking .... continuing with system checks.

Generating detailed listing and cross-reference of the input document.

Re-generating the original input document.

Knowledge base checking completed.

Do you want a system model developed for this knowledge base? y

The system will now generate a system model based on the data contained within the initial English prose requirements document.

Generating system model from the system knowledge base ... done.

Prototyping completed.
Knowledge Base Facts

The knowledge base constructed during environment generation is illustrated.

The knowledge base uses a PROLOG-based syntax for presentation of these facts.

```
sentence( s_001 , [the,widget,warehouse,company,sells,products, to,its,customers] ).
object( o_001 , [widget,warehouse,company] ).
object( o_002 , [product] ).
action( a_001 , [sell] ).
object( o_003 , [customer] ).
restriction( r_001 , object , to ).
r_object( r_001 , o_003 , r_0 ).
event( e_001 ).
action( e_001 , a_001 , r_001 ).
actor( e_001 , o_001 , r_0 ).
object( e_001 , o_002 , r_0 ).
generator( e_001 , s_001 ).

sentence( s_002 , [the,company,has,a,clerk,who,handles,the,customers, and,their/orders] ).
object( o_004 , [clerk] ).
object( o_005 , [order] ).
action( a_002 , [handle] ).
event( e_002 ).
action( e_002 , a_002 , r_0 ).
actor( e_002 , o_002 , r_0 ).
object( e_002 , o_004 , r_0 ).
event( e_003 ).
action( e_003 , a_002 , r_0 ).
actor( e_003 , o_004 , r_0 ).
object( e_003 , o_003 , r_0 ).
restriction( r_002 , event , who ).
r_event( r_002 , e_003 ).
property( o_001 , object , p_001 ).
p_restrict( p_001 , r_0 ).
p_object( p_001 , o_004 , r_002 ).
generator( e_002 , s_002 ).
generator( e_003 , s_002 ).
generator( p_001 , s_002 ).
```
sentence( s_003 , [customers,order,these,products,by,phone] ).
action( a_003 , [order] ).
object( o_006 , [phone] ).
restriction( r_003 , object , by ).
r_object( r_003 , o_006 , r_0 ).
event( e_004 ).
action( e_004 , a_003 , r_003 ).
actor( e_004 , o_003 , r_0 ).
object( e_004 , o_002 , r_0 ).
generator( e_004 , s_003 ).

____________________________________

sentence( s_004 , [each,order,has,a,customer,name,xxx,customer,address,xxx,
product,id,xxx,quantity,and,delivery,date] ).
object( o_007 , [customer,name] ).
object( o_008 , [customer,address] ).
object( o_009 , [product,id] ).
object( o_010 , [quantity] ).
object( o_011 , [delivery,date] ).
property( o_005 , object , p_002 ).
p_restrict( p_002 , r_0 ).
p_object( p_002 , o_011 , r_0 ).
property( o_005 , object , p_003 ).
p_restrict( p_003 , r_0 ).
p_object( p_003 , o_010 , r_0 ).
property( o_005 , object , p_004 ).
p_restrict( p_004 , r_0 ).
p_object( p_004 , o_009 , r_0 ).
property( o_005 , object , p_005 ).
p_restrict( p_005 , r_0 ).
p_object( p_005 , o_008 , r_0 ).
property( o_005 , object , p_006 ).
p_restrict( p_006 , r_0 ).
p_object( p_006 , o_007 , r_0 ).
generator( p_002 , s_004 ).
generator( p_003 , s_004 ).
generator( p_004 , s_004 ).
generator( p_005 , s_004 ).
generator( p_006 , s_004 ).

____________________________________

sentence( s_005 , [the,clerk,allocates,stock,and,ships,the,orders] ).
object( o_012 , [stock] ).
action( a_004 , [allocate] ).
action( a_005 , [ship] ).
event( e_005 ).
action( e_005 , a_004 , r_0 ).
actor( e_005 , o_004 , r_0 ).
object( e_005 , o_012 , r_0 ).
event( e_006 ).
action( e_006 , a_005 , r_0 ).
actor( e_006, o_004, r_0 ).
object( e_006, o_005, r_0 ).
generator( e_005, s_005 ).
generator( e_006, s_005 ).

text( s_006, [the, clerk, bills, the, customer, with, the, shipment] ).
action( a_006, [bill] ).
object( o_013, [shipment] ).
restriction( r_004, object, with ).
r_object( r_004, o_013, r_0 ).
event( e_007 ).
action( e_007, a_006, r_004 ).
actor( e_007, o_004, r_0 ).
object( e_007, o_003, r_0 ).
generator( e_007, s_006 ).

text( s_007, [the, clerk, maintains, the, current, inventory, and, purchases, 'out-of-stock', products] ).
object( o_014, [inventory] ).
action( a_007, [maintain] ).
action( a_008, [purchase] ).
quantifier( current, q_001, english ).
q_limits( q_001, ['the', 'amount', 'in', 'stock', 'at', 'the', 'start', 'of', 'the', 'day'] ).
restriction( r_005, simple, current ).
r_simple( r_005, q_001 ).
event( e_008 ).
action( e_008, a_007, r_0 ).
actor( e_008, o_004, r_0 ).
object( e_008, o_014, r_005 ).
restriction( r_006, simple, 'out-of-stock' ).
r_simple( r_006, q_0 ).
event( e_009 ).
action( e_009, a_008, r_0 ).
actor( e_009, o_004, r_0 ).
object( e_009, o_002, r_006 ).
generator( e_008, s_007 ).
generator( e_009, s_007 ).

text( s_008, [each, product, has, a, product, id, xxx, inventory, price, and, customer, price] ).
object( o_015, [inventory, price] ).
object( o_016, [customer, price] ).
property( o_002, object, p_007 ).
p_restrict( p_007, r_0 ).
p_object( p_007, o_016, r_0 ).
property( o_002, object, p_008 ).
p_restrict( p_008, r_0 ).
Document Report

The data within the knowledge base is evaluated for reliability, testability and traceability. A document report is generated by this analysis procedure. It contains a listing of the document, the errors and warnings found within the document, as well as a cross-reference of the document by sentence, object and action. This document is
designed to illustrate the complete specification paradigm, from initial document parsing to the generation of the specification model. An error list is not generated in this example, since the errors within the document have been removed during prior applications of the specification paradigm. Therefore, only the document listing and cross-references generated by the evaluation process are illustrated.

Document Listing

s_001 : the widget warehouse company sells products to its customers
s_002 : the company has a clerk who handles the customers and their orders
s_003 : customers order these products by phone
s_004 : each order has a customer name, customer address, product id, quantity and delivery date
s_005 : the clerk allocates stock and ships the orders
s_006 : the clerk bills the customer with the shipment
s_007 : the clerk maintains the current inventory and purchases out-of-stock products
s_008 : each product has a product id, inventory price and customer price
s_009 : the customer can modify the quantity and delivery date
s_010 : the customer may cancel their current order

Cross Reference Listing

Listing By Sentence

Sentence: s_001 the widget warehouse company sells products to its customers

Objects used:
- customer
- restriction to action sell
- product
- object of action sell
- widget warehouse company
- actor of action sell

Actions used:
- sell
Sentence: s_002  the company has a clerk who handles the customers and their orders
Objects used:
  clerk
    actor of action handle
  customer
    object property of widget warehouse company
  order
Actions used:
  handle

Sentence: s_003  customers order these products by phone
Objects used:
  customer
    actor of action order
  phone
    restriction to action order
  product
    object of action order
Actions used:
  order

Sentence: s_004  each order has a customer name, customer address, product id, quantity and delivery date
Objects used:
  customer address
    object property of order
  customer name
    object property of order
  delivery date
    object property of order
  order
    object property defined for this entity
  product id
    object property of order
  quantity
    object property of order
Actions used:
  ** NONE **
Sentence: s_005  the clerk allocates stock and ships the orders
Objects used:
clerk
  actor of action allocate
  actor of action ship
order
  object of action ship
stock
  object of action allocate
Actions used:
  allocate
  ship

Sentence: s_006  the clerk bills the customer with the shipment
Objects used:
clerk
  actor of action bill
customer
  object of action bill
shipment
  restriction to action bill
Actions used:
  bill

Sentence: s_007  the clerk maintains the current inventory and purchases out-of-stock products
Objects used:
clerk
  actor of action maintain
  actor of action purchase
inventory
  object of action maintain
product
  object of action purchase
Actions used:
  maintain
  purchase
Sentence: s_008  each product has a product id, inventory price and customer price
Objects used:
customer price
  object property of product
inventory price
  object property of product
product
  object property defined for this entity
product id
  object property of product
Actions used:
** NONE **

Sentence: s_009  the customer can modify the quantity and delivery date
Objects used:
customer
  actor of action modify
delivery date
  object of action modify
quantity
  object of action modify
Actions used:
modify

Sentence: s_010  the customer may cancel their current order
Objects used:
customer
  actor of action cancel
order
  object of action cancel
Actions used:
cancel
Listing By Object

clerk
Operations include:
  s_002 : actor of action handle
  s_005 : actor of action allocate
  s_005 : actor of action ship
  s_006 : actor of action bill
  s_007 : actor of action maintain
  s_007 : actor of action purchase
Attributes defined in:
  s_002 : object property of the entity widget warehouse company
No hierarchy ordering defined.

customer
Operations include:
  s_003 : actor of action order
  s_009 : actor of action modify
  s_010 : actor of action cancel
  s_002 : object of action handle
  s_006 : object of action bill
No attributes defined.
No hierarchy ordering defined.

customer address
  Does not participate in any operations.
Attributes defined in:
  s_004 : object property of the entity order
No hierarchy ordering defined.

customer name
  Does not participate in any operations.
Attributes defined in:
  s_004 : object property of the entity order
No hierarchy ordering defined.

customer price
  Does not participate in any operations.
Attributes defined in:
  s_008 : object property of the entity product
No hierarchy ordering defined.
delivery date
  Operations include:
    s_009 : object of action modify
  Attributes defined in:
    s_004 : object property of the entity order
  No hierarchy ordering defined.

inventory
  Operations include:
    s_007 : object of action maintain
  No attributes defined.
  No hierarchy ordering defined.

inventory price
  Does not participate in any operations.
  Attributes defined in:
    s_008 : object property of the entity product
  No hierarchy ordering defined.

order
  Operations include:
    s_002 : object of action handle
    s_005 : object of action ship
    s_010 : object of action cancel
  Attributes defined in:
    s_004 : object property defined for this entity
  No hierarchy ordering defined.

phone
  Does not participate in any operations.
  No attributes defined.
  No hierarchy ordering defined.

product
  Operations include:
    s_001 : object of action sell
    s_003 : object of action order
    s_007 : object of action purchase
  Attributes defined in:
    s_008 : object property defined for this entity
  No hierarchy ordering defined.
product id
Does not participate in any operations.
Attributes defined in :
\( s_{004} \) : object property of the entity order
\( s_{008} \) : object property of the entity product
No hierarchy ordering defined.

quantity
Operations include :
\( s_{009} \) : object of action modify
Attributes defined in :
\( s_{004} \) : object property of the entity order
No hierarchy ordering defined.

shipment
Does not participate in any operations.
No attributes defined.
No hierarchy ordering defined.

stock
Operations include :
\( s_{005} \) : object of action allocate
No attributes defined.
No hierarchy ordering defined.

widget warehouse company
Operations include :
\( s_{001} \) : actor of action sell
Attributes defined in :
\( s_{002} \) : object property defined for this entity
No hierarchy ordering defined.

Listing By Action

allocate
Used in sentences : \( s_{005} \)

bill
Used in sentences : \( s_{006} \)

cancel
Used in sentences : \( s_{010} \)
Regenerated Specification Document

The information within the initial requirements document is restructured into an object-based specification document. This document provides an organized presentation of the document’s information.

INITIAL INPUT REQUIREMENTS DOCUMENT

The widget warehouse company sells products to its customers. The company has a clerk who handles the customers and their orders. Customers order these products by phone. Each order has a customer name, customer address, product id, quantity and delivery date. The clerk allocates stock and ships the orders. The clerk bills the customer with the shipment. The clerk maintains the current inventory and purchases out-of-stock products. Each product has a product id, inventory price and customer price. The customer can modify the quantity and delivery date. The customer may cancel their current order.
The following is a list of all logical objects referenced in this system:

- clerk
- customer
- customer address
- customer name
- customer price
- delivery date
- inventory
- inventory price
- order
- product
- product id
- quantity
- stock
- widget warehouse company

A detailed description of the objects in the system is now presented. The objects are listed in alphabetical order. This data includes:

1. A complete listing of the known hierarchy relationships between it and all other objects in the system.
2. All properties, both physical and event-oriented, that are characteristic of that object.
3. A listing of all events with which this object is associated.
4. A description of the object visibility. This is a listing of all other objects that act upon the given object, requiring that this object be accessible to them.

clerk

No hierarchy relationships exist between clerk and any other objects within the system.

There are no physical or event properties defined for clerk.
clerk is represented as a property in:

- widget warehouse company has clerk who handle customer

clerk instigates the following events within the system:

- clerk handle order
- clerk handle customer
- clerk allocate stock
- clerk ship order
- clerk bill customer with shipment
- clerk maintain current [the amount in stock at the start of the day] inventory
- clerk purchase out-of-stock product

clerk is not effected by any actions within the system.
clerk does not benefit from any actions within the system.
clerk is not visible to any other object within the system.
clerk is found in the original document in the sentences:

s_002: the company has a clerk who handles the customers and their orders
s_005: the clerk allocates stock and ships the orders
s_006: the clerk bills the customer with the shipment
s_007: the clerk maintains the current inventory and purchases out-of-stock products

customer
No hierarchy relationships exist between customer and any other objects within the system.
There are no physical or event properties defined for customer.
customer instigates the following events within the system:
customer order product by phone
customer modify delivery date
customer modify quantity
customer cancel current [ any outstanding unfilled order ] order
customer is acted upon in the following events within the system:
clerk handle customer
clerk bill customer with shipment
customer does not benefit from any actions within the system.
customer is visible to the following objects within the system:
visible to 'clerk' through the operation 'handle'
visible to 'clerk' through the operation 'bill'
customer is found in the original document in the sentences:
s_001: the widget warehouse company sells products to its customers
s_002: the company has a clerk who handles the customers and their orders
s_003: customers order these products by phone
s_006: the clerk bills the customer with the shipment
s_009: the customer can modify the quantity and delivery date
s_010: the customer may cancel their current order

customer address
No hierarchy relationships exist between customer address and any other objects within the system.
There are no physical or event properties defined for customer address.
customer address is represented as a property in:
order has customer address
customer address does not actively participate in any actions within the system.
customer address is not visible to any other object within the system.
customer address is found in the original document in the sentences:

customer name
No hierarchy relationships exist between customer name and any other objects within the system.
There are no physical or event properties defined for customer name.
customer name is represented as a property in:
   order has customer name
customer name does not actively participate in any actions within the system.
customer name is not visible to any other object within the system.
customer name is found in the original document in the sentences:

**customer price**
No hierarchy relationships exist between customer price and any other objects within the system.
There are no physical or event properties defined for customer price.
customer price is represented as a property in:
   product has customer price
customer price does not actively participate in any actions within the system.
customer price is not visible to any other object within the system.
customer price is found in the original document in the sentences:

**delivery date**
No hierarchy relationships exist between delivery date and any other objects within the system.
There are no physical or event properties defined for delivery date.
delivery date is represented as a property in:
   order has delivery date
delivery date does not instigate any actions within the system.
delivery date is acted upon in the following events within the system:
   customer modify delivery date
delivery date does not benefit from any actions within the system.
delivery date is visible to the following objects within the system:
   visible to 'customer' through the operation 'modify'
delivery date is found in the original document in the sentences:
   s_009 : the customer can modify the quantity and delivery date

**inventory**
No hierarchy relationships exist between inventory and any other objects within the system.
There are no physical or event properties defined for inventory.
inventory does not instigate any actions within the system.
inventory is acted upon in the following events within the system:
   clerk maintain current [ the amount in stock at the start of the day ] inventory
inventory does not benefit from any actions within the system.
inventory is visible to the following objects within the system:
   visible to 'clerk' through the operation 'maintain'
inventory is found in the original document in the sentences:
   s_007 : the clerk maintains the current inventory and purchases out-of-stock products
**inventory price**

No hierarchy relationships exist between inventory price and any other objects within the system.

There are no physical or event properties defined for inventory price.

inventory price is represented as a property in:

- product has inventory price

inventory price does not actively participate in any actions within the system.

inventory price is not visible to any other object within the system.

inventory price is found in the original document in the sentences:

**order**

No hierarchy relationships exist between order and any other objects within the system.

order exhibits the following properties:

- order has delivery date
- order has quantity
- order has product id
- order has customer address
- order has customer name

order does not instigate any actions within the system.

order is acted upon in the following events within the system:

- clerk handle order
- clerk ship order
- customer cancel current [ any outstanding unfilled order ] order

order does not benefit from any actions within the system.

order is visible to the following objects within the system:

- visible to 'clerk' through the operation 'handle'
- visible to 'clerk' through the operation 'ship'
- visible to 'customer' through the operation 'cancel'

order is found in the original document in the sentences:

- s_002: the company has a clerk who handles the customers and their orders
- s_004: each order has a customer name, customer address, product id, quantity and delivery date
- s_005: the clerk allocates stock and ships the orders
- s_010: the customer may cancel their current order

**product**

No hierarchy relationships exist between product and any other objects within the system.

product exhibits the following properties:

- product has customer price
- product has inventory price
- product has product id

product does not instigate any actions within the system.
product is acted upon in the following events within the system:
- widget warehouse company sell product to customer
- customer order product by phone
- clerk purchase out-of-stock product

product does not benefit from any actions within the system.

product is visible to the following objects within the system:
- visible to 'widget warehouse company' through the operation 'sell'
- visible to 'customer' through the operation 'order'
- visible to 'clerk' through the operation 'purchase'

product is found in the original document in the sentences:
- s_001: the widget warehouse company sells products to its customers
- s_003: customers order these products by phone
- s_007: the clerk maintains the current inventory and purchases out-of-stock products
- s_008: each product has a product id, inventory price and customer price

**product id**

No hierarchy relationships exist between product id and any other objects within the system.

There are no physical or event properties defined for product id.

product id is represented as a property in:
- order has product id
- product has product id

product id does not actively participate in any actions within the system.

product id is not visible to any other object within the system.

product id is found in the original document in the sentences:

**quantity**

No hierarchy relationships exist between quantity and any other objects within the system.

There are no physical or event properties defined for quantity.

quantity is represented as a property in:
- order has quantity

quantity does not instigate any actions within the system.

quantity is acted upon in the following events within the system:
- customer modify quantity

quantity does not benefit from any actions within the system.

quantity is visible to the following objects within the system:
- visible to 'customer' through the operation 'modify'

quantity is found in the original document in the sentences:
- s_009: the customer can modify the quantity and delivery date

**stock**

No hierarchy relationships exist between stock and any other objects within the system.
There are no physical or event properties defined for stock. stock does not instigate any actions within the system. stock is acted upon in the following events within the system :
   clerk allocate stock
stock does not benefit from any actions within the system. stock is visible to the following objects within the system :
   visible to 'clerk' through the operation 'allocate'
stock is found in the original document in the sentences :
   s_005 : the clerk allocates stock and ships the orders

widget warehouse company
No hierarchy relationships exist between widget warehouse company and any other objects within the system.
widget warehouse company exhibits the following properties :
   widget warehouse company has clerk who handle customer
widget warehouse company instigates the following events within the system :
   widget warehouse company sell product to customer
widget warehouse company is not effected by any actions within the system.
widget warehouse company does not benefit from any actions within the system.
widget warehouse company is not visible to any other object within the system.
widget warehouse company is found in the original document in the sentences :
   s_001 : the widget warehouse company sells products to its customers
   s_002 : the company has a clerk who handles the customers and their orders

---

**Generated Specification Model**

The object-oriented specification paradigm utilizes the information contained within the generated specification environment’s knowledge base to create the following specification model. This model contains an object-oriented specification for the system, as it is defined within the initial requirements document.
package customer
model Abstract State Machine
requires order, product
contains <>
procedure order [ product:external ]
   -- customer order product by phone
procedure modify [ delivery date:external ]
   -- customer modify delivery date
procedure modify [ quantity:external ]
   -- customer modify quantity
procedure cancel [ order:external ]
   -- customer cancel current [ any outstanding unfilled order ] order
end customer

package order
model Abstract Data Type
requires <>
contains <>
property delivery date:object
   -- order has delivery date
property quantity:object
   -- order has quantity
property product id:object
   -- order has product id
property customer address:object
   -- order has customer address
property customer name:object
   -- order has customer name
end order
### Product Package

<table>
<thead>
<tr>
<th>Model</th>
<th>Abstract Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>Contains</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>Property</td>
<td>customer price:object</td>
</tr>
<tr>
<td></td>
<td>-- product has customer price</td>
</tr>
<tr>
<td>Property</td>
<td>inventory price:object</td>
</tr>
<tr>
<td></td>
<td>-- product has inventory price</td>
</tr>
<tr>
<td>Property</td>
<td>product id:object</td>
</tr>
<tr>
<td></td>
<td>-- product has product id</td>
</tr>
</tbody>
</table>

### Widget Warehouse Company Package

<table>
<thead>
<tr>
<th>Model</th>
<th>Abstract State Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires</td>
<td>product</td>
</tr>
<tr>
<td>Contains</td>
<td>clerk</td>
</tr>
<tr>
<td>Property</td>
<td>clerk:object is separate</td>
</tr>
<tr>
<td></td>
<td>-- widget warehouse company has clerk who handle customer</td>
</tr>
<tr>
<td>Procedure</td>
<td>sell [ product:external ]</td>
</tr>
<tr>
<td></td>
<td>-- widget warehouse company sell product to customer</td>
</tr>
</tbody>
</table>

end product

end widget warehouse company
package-part clerk belongs to widget warehouse company

model Abstract State Machine

requires customer, product, order

contains <>

procedure handle [ order:external ]
  -- clerk handle order

procedure handle [ customer:external ]
  -- clerk handle customer

procedure allocate [ stock:internal ]
  -- clerk allocate stock

procedure ship [ order:external ]
  -- clerk ship order

procedure bill [ customer:external ]
  -- clerk bill customer with shipment

procedure maintain [ inventory:internal ]
  -- clerk maintain current [ the amount in stock at the start of the day ] inventory

procedure purchase [ product:external ]
  -- clerk purchase out-of-stock product

internal stock

internal inventory

end clerk
Vita

David W. Cordes has been an Alumni Federation Fellow at Louisiana State University since August of 1985. His current research areas include software engineering, with emphasis towards requirements specification development. Other interests include programming languages, compilers and operating systems. He is responsible for publishing the Department’s monthly newsletter and also assists in the operation and maintenance of the Departmental UNIX machines.

Prior to enrolling in LSU, he did graduate work at Purdue University, where he received his Master’s degree in the Spring of 1984. He completed his undergraduate work at the University of Arkansas, graduating with a B.S. in Computer Science in May of 1982.

After graduation, he is taking a position as Assistant Professor in the Department of Computer Science at the University of Alabama.
Candidate: David W. Cordes

Major Field: Computer Science

Title of Dissertation: An Object-Oriented Paradigm for Requirements Specifications

Approved:

[Signatures]

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

[Signatures]

Date of Examination:

July 5, 1988