

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A212 085

DTIC
ELECTE
SEP 07 1989
S D



THESIS

THE DESIGN AND IMPLEMENTATION OF A
SYNTAX DIRECTED EDITOR FOR THE
SPECIFICATION LANGUAGE SPEC

by

David Harry Beebe

June 1989

Thesis Advisor:

Valdis Berzins

Approved for public release; distribution is unlimited.

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified			1b Restrictive Markings		
2a Security Classification Authority			3 Distribution Availability of Report		
2b Declassification/Downgrading Schedule			Approved for public release; distribution is unlimited.		
4 Performing Organization Report Number(s)			5 Monitoring Organization Report Number(s)		
6a Name of Performing Organization Naval Postgraduate School		6b Office Symbol (If Applicable) 52	7a Name of Monitoring Organization Naval Postgraduate School		
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000			7b Address (city, state, and ZIP code) Monterey, CA 93943-5000		
8a Name of Funding/Sponsoring Organization		8b Office Symbol (If Applicable)	9 Procurement Instrument Identification Number		
8c Address (city, state, and ZIP code)			10 Source of Funding Numbers		
			Program Element Number	Project No	Task No
			Work Unit Accession No		
11 Title (Include Security Classification) THE DESIGN AND IMPLEMENTATION OF A SYNTAX DIRECTED EDITOR FOR THE SPECIFICATION LANGUAGE SPEC					
12 Personal Author(s) Beebe, David H.					
13a Type of Report Master's Thesis		13b Time Covered From To		14 Date of Report (year, month, day) June 1989	
15 Page Count 164					
16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.					
17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)		
Field	Group	Subgroup	Syntax Directed Editor, Spec; Specification Language; Attribute Grammar;		
			Abstract Syntax; Concrete Syntax; Synthesizer Generator		
19 Abstract (continue on reverse if necessary and identify by block number) The formal specification language Spec is used for writing black-box specifications for large software systems. These black-box specifications describe the interface between a system and its users, as well as internal interfaces between modules. Systems analysts use specifications written in Spec to verify the customer's requirements during the development of a software system. This thesis demonstrates the feasibility of designing and implementing a syntax directed editor for a subset of the specification language Spec. The editor is a software tool for writing Spec specifications that ensures syntactic correctness of such specifications. The syntax directed editor is created using the Synthesizer Generator, a Computer-Aided Software Engineering (CASE) tool for generating language-based editors. The specification for the editor is written in the Synthesizer Specification Language (SSL) which is based on an attribute grammar. The software tool developed in this thesis supports the Requirements Analysis phase of the software development cycle.					
20 Distribution/Availability of Abstract ☒ unclassified/unlimited ☐ same as report ☐ DTIC users			21 Abstract Security Classification Unclassified		
22a Name of Responsible Individual Prof. Valdis A. Berzins			22b Telephone (Include Area code) (408) 646-2461		22c Office Symbol Code 52Bz

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted

security classification of this page

All other editions are obsolete

Unclassified

Approved for public release; distribution is unlimited.

**The Design and Implementation of a Syntax Directed Editor
for the Specification Language Spec**

by

David Harry Beebe
Lieutenant Commander, United States Navy
B.A., Western Michigan University, 1975

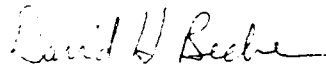
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
JUNE 1989**

Author:



David Harry Beebe

Approved by:



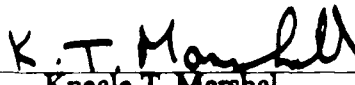
Valdis Berzins, Thesis Advisor



Luqi, Second Reader



Robert McGhee, Chairman, Department of
Computer Science



Kneale T. Marshall
Dean of Information and Policy Science

ABSTRACT

The formal specification language Spec is used for writing black-box specifications for large software systems. These black-box specifications describe the interface between a system and its users, as well as internal interfaces between modules. Systems analysts use specifications written in Spec to verify the customer's requirements during the development of a software system.

This thesis demonstrates the feasibility of designing and implementing a syntax directed editor for a subset of the specification language Spec. The editor is a software tool for writing Spec specifications that ensures syntactic correctness of such specifications. The syntax directed editor is created using the Synthesizer Generator, a Computer-Aided Software Engineering (CASE) tool for generating language-based editors. The specification for the editor is written in the Synthesizer Specification Language (SSL) which is based on an attribute grammar. The software tool developed in this thesis supports the Requirements Analysis phase of the software development cycle.

Collection For	
THIS CHART	☒
TABLE	☐
FIGURE	☐
Justification	
By	
Distribution	
Availability Codes	
Dist	Availability Codes
Special	
A-1	



TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	OVERVIEW	1
B.	DESCRIPTION OF SOFTWARE ENGINEERING PRINCIPLES	2
C.	COMPUTER-AIDED SOFTWARE ENGINEERING (CASE)	6
D.	FOURTH-GENERATION LANGUAGES	7
II.	BACKGROUND	10
A.	GENERAL DESCRIPTION OF A SYNTAX DIRECTED EDITOR	10
B.	SYNTAX DIRECTED EDITOR GENERATORS	13
1.	The GANDALF System	14
2.	The Synthesizer Generator	16
III.	AN OVERVIEW OF THE SPEC SPECIFICATION LANGUAGE	19
A.	THE PURPOSE OF SPEC	19
B.	MODULE DESCRIPTIONS	20
1.	Definition Modules	20
2.	Function Modules	21
3.	Machine Modules	22
4.	Type Modules	24
5.	Instance Modules	25
C.	OTHER FEATURES	25
1.	Concepts used as Subprograms	25
2.	Inheritance in Spec	26
D.	COMPLETE DESCRIPTION OF SPEC	27
IV.	DESIGN OF THE SYNTAX DIRECTED EDITOR FOR SPEC	28
A.	GOALS	28
1.	Short Term	28
2.	Long Term	28
B.	CONCEPTS FOR USING THE SYNTHESIZER GENERATOR	28
1.	Attribute Grammar	29
2.	Abstract Syntax, Phyla and Terms	30
3.	Completing Term	32
4.	Placeholder Term	32
5.	Lists and Optional Lists	33
6.	Optional (Non-List) Phyla	36
7.	Lexical Phyla	36

C. PRACTICAL APPROACH TO DESIGNING AN EDITOR	39
1. Define the Abstract Syntax	40
2. Define a Display Representation	41
3. Define Template Transformations	46
4. Generate and Debug Editor	48
5. Define Concrete Input Syntax	48
a. Parsing Declarations	49
b. Correspondence Between Concrete Syntax and Abstract Syntax	52
c. Concrete Lexical Declarations	54
d. Parsing and Ambiguities	55
6. Refine the Display Representation	59
7. Advanced Unparsing Features	61
D. DESIGN DECISIONS	63
1. Spec Subset	63
2. Type Checking	63
3. Display Representation	64
4. Naming Conventions	64
5. Implementation	65
V. CONCLUSIONS	66
A. APPLICABILITY OF THE SYNTAX DIRECTED EDITOR FOR SPEC	66
B. USEFULNESS OF THE SYNTHESIZER GENERATOR	67
C. DEFICIENCIES AND BUGS ENCOUNTERED	69
D. RECOMMENDATIONS FOR FUTURE RESEARCH	69
APPENDIX A - COMPLETE SPEC GRAMMAR	72
APPENDIX B - SPECDEF SSL SPECIFICATION FILES	87
APPENDIX C - USER'S MANUAL FOR THE SPECDEF EDITOR	105
APPENDIX D - LIST OF EDITOR COMMANDS	150
REFERENCES	153
INITIAL DISTRIBUTION LIST	155

ACKNOWLEDGEMENT

I would like to thank Professor Valdis Berzins and Professor Luqi for stimulating my interest in the software engineering discipline of computer science. Their help and words of encouragement got me over seemingly insurmountable hurdles and kept me going when it seemed hopeless.

I would also like to thank Lieutenant Robert Kopas, U. S. Navy, for his tirelessness in explaining the concepts of attribute grammars, lexical analysis, parsing, and the like, and for proofreading my work and offering helpful comments and suggestions. His enthusiasm and energy were a constant source of inspiration.

A very special thank you to my wife Jan and daughter Ruth for their love, patience and endurance during these past months. Without their family support, I might not have survived the ordeal unscathed.

I. INTRODUCTION

A. OVERVIEW

Spec is a formal specification language designed to write black-box specifications of software systems. These black-box specifications describe the interface between a system and its users as well as internal interfaces between modules. Specifications written in Spec are used by systems analysts to verify the customer's requirements. The specification is then translated into an appropriate programming language (such as Ada) by programmers.

Currently, Spec is a research tool with the potential to become an industry standard in software development. An eventual goal is to create an integrated environment of software tools for Spec. These tools would include an editor, type checker, consistency checker, inheritance expander, pretty printer, test oracle, diagram generator and a translator to Ada.

A syntax directed editor is the primary tool of such an integrated environment. The user is guaranteed a syntactically correct Spec specification at the end of each editing session. By ensuring syntactic correctness as the specification is being designed/edited, the overall time spent designing the system is reduced.

This thesis discusses the development of SPECDEF, a syntax directed editor for a subset of the Spec specification language. Chapter II gives a general description of a syntax directed editor and compares two primary editor generator systems. Chapter III is

a description of the Spec language. The complete Spec grammar appears in Appendix A. Chapter IV details the concepts behind the Synthesizer Generator, design decisions, and a practical approach to designing a syntax directed editor specification. The specification files for the SPECDEF editor appear in Appendix B. A user's manual for the SPECDEF editor appears in Appendix C, and a listing of all editor system commands appears in Appendix D.

B. DESCRIPTION OF SOFTWARE ENGINEERING PRINCIPLES

Computers are playing an increasingly large role in our lives. Many manufacturing processes are being automated. retail and grocery stores have computerized pricing and inventory systems. banks are providing automatic teller machines, travel agents make our plane reservations through computers, etc. In order for these and other computer systems to perform useful tasks, reliable and efficient software is required.

Software engineering applies scientific and mathematical principles to the development of software in order to make computers useful to people. Software consists of the actual programs, an documentation and user's guides, operating procedures and test cases that are associated with a computer-based system. A primary goal of software engineering is the development of effective scientific methods for producing software that meets the customer's schedule and budget constraints while satisfying all his requirements for the software system. [Ref. 1]

Computers can perform tasks that are repetitious, too time consuming, or too complicated for people to do manually, and they can usually perform those tasks faster, with greater reliability, more efficiently and at a lower cost than people can. In order for

computers to be useful, the software system designed for the customer must be able to perform all functions required by the user in a correct and efficient manner. Any failure of the system to operate correctly could be costly in terms of lives, equipment, and money. Software systems should not only be "correct", i.e., conform to the specification, but also should perform the functions desired by the customer. Correct but inappropriate systems are sometimes built because the software developers did not understand the user's needs, or the software cannot adapt to changes in those needs. [Ref. 1]

Hardware prices have decreased by about 50% roughly every two years. This in turn has triggered a demand for larger, more sophisticated computer applications. It is impossible for a single person to understand or to build large and complex software systems. Therefore, a software development organization is needed. The system must be organized as a set of modules that are small enough to be developed by a single person. Each of the developers in the organization must communicate with the other developers, either directly or through documentation. This documentation must be precise so that all developers on the project team have a complete and correct understanding of the interface between the module(s) they are developing. In addition to this formal documentation, communication is extremely important, particularly in large projects, where project members come and go. [Ref. 1]

The software development process consists of a cycle of qualitative activities: requirements analysis, functional specification, architectural design, implementation, and evolution. The development will proceed most efficiently if these activities are performed in a pipeline fashion. However, each activity does not necessarily end when the

next begins. Often, insights are gained at later stages that trigger modifications or extensions to earlier results. Therefore, feedback is essential. The relationship between activities of the software development cycle is illustrated in Figure 1.

Requirements analysis is the process of determining and documenting the customer's needs. The purpose of the system, as well as any constraints on its development, is determined at this stage. A systems interface is proposed and formalized in the functional specification stage. This interface only describes those aspects of the system behavior that are visible to the user or to other external systems. The system is then decomposed into modules in the architectural design phase. Internal interfaces, i.e., those not visible to the user, are defined here. Implementation is the production of a program for each of the modules. The data structures and algorithms used within each module are defined here. Finally, the evolution, or repair process, allows for adaptation of the system to changing needs of the customer. Any discrepancies between the specification and the implementation that are discovered after system delivery are repaired here, and the capabilities of the system are expanded as new requirements are discovered. This evolution requires repeating the previous four steps of the software development cycle.

[Ref. 1]

The SPECDEF editor will be useful mostly in the requirements analysis phase of the software development cycle, and to a lesser extent in the functional specification and architectural design phases.

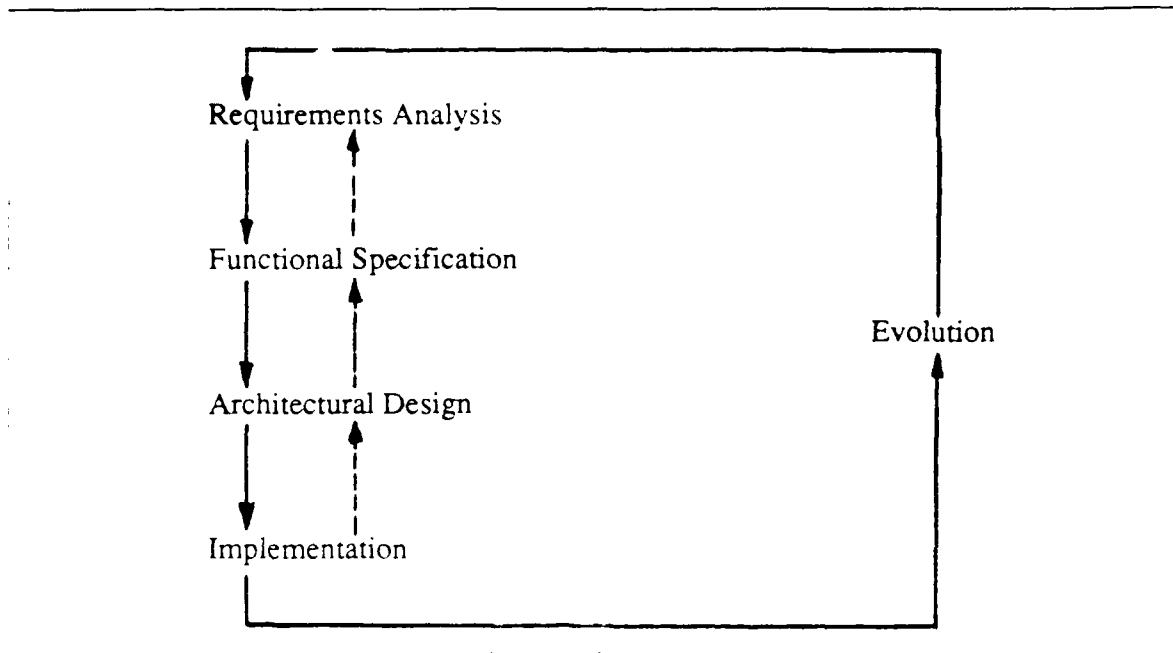


Figure 1
Software Development Cycle

Thorough analysis and testing of each module is absolutely necessary if a "correct" system is to be developed. A project database should be utilized to record and distribute all information about the state of the development process. This database will contain a variety of documents, including the requirements, designs, justifications for those designs, the code, test cases and results, user's manuals, the schedule of the development project, work assignments, etc. The larger the system, the larger the database, and the greater the need for a database management system. For engineering applications, a specialized engineering database system is required to efficiently manage the project database. This is a relatively new development, and mature systems are not yet widely available. Future versions of SPECDEF should interface to this design database. [Ref. 1]

C. COMPUTER-AIDED SOFTWARE ENGINEERING (CASE)

The time spent in the requirements analysis, functional specification and architectural design stages of the software development cycle has been greatly reduced by the advancements made in computer-aided software engineering (CASE) tools. These tools assist the systems analyst and software engineer in specifying the system's requirements and design. Many of the design and development problems inherent in medium to large software projects are reduced or eliminated by the use of CASE tools. [Ref. 2]

Some CASE tools can automatically generate code from the software design specification. Two benefits are achieved from this. First, the implementation time is substantially reduced since the code (or at least part of it) is being generated automatically. Secondly, the software engineer may feel more confident about the quality of the generated code because it was produced by a software tool that has itself been thoroughly tested and debugged. [Ref. 2]

Truly generalized code generation is not available in any of today's general-purpose tools. However, specialty development tools that focus on a particular type of software, such as user interface design tools, are available that will generate code. This thesis will discuss the use of one such tool, the Synthesizer Generator, which generates a syntax directed editor program. This generated program is in fact another example of a CASE tool.

In general, CASE tools are designed to increase the productivity of systems analysts and software engineers. Many of these tools are actually requirements and design specification editors that provide output in specific formats, often graphically oriented,

that can be read and understood by end-users and developers alike. They eliminate the drudgery of drawing and redrawing data flow diagrams, module hierarchy charts, etc., that often change as the project development proceeds. Time is saved and formats are consistent. Future versions of the SPECDEF editor should interface with tools for generating such diagrams and other summary information. [Ref. 2]

D. FOURTH-GENERATION LANGUAGES

The term "fourth-generation language" can be looked at in two different ways. Some authors use the term to refer to application generator programs. Applications generators are software tools that take a design specification as input and produce compilable code as output. When used in this context, the "fourth-generation language" is not always a programming language in the sense that a programming language is intended to describe a program [Ref. 3]. This context includes many CASE tools, such as program generators. Fourth-generation languages in this context are sometimes thought of as higher-level focused languages that provide mechanisms for accessing data bases [Ref. 2]. Many software professionals think of fourth-generation languages as a form of CASE [Ref. 2].

Other authors, when referring to "fourth-generation languages," mean a particular class of modern *programming* languages, including Ada. Fourth-generation programming languages provide mechanisms for *data abstraction*. An encapsulation facility supports the separation of the specification and definition of data structures, *information hiding*, and *name access by mutual consent*. A facility for passing messages between concurrent tasks to maintain synchronization and communication supports concurrent programming. [Ref. 3]

A software crisis developed in the mid 1970's because software development had become labor intensive rather than a labor-saving activity. There were problems with responsiveness of computer systems to customers' needs, reliability of software, escalating software development costs, maintainability of software systems, timeliness of development of software systems, transportability of software from one operating environment to another, and efficiency of the software in terms of processing time and memory space. Software systems were becoming increasingly larger and more complex. The most popular programming languages at that time, FORTRAN and COBOL, did not reflect newly developed software engineering principles and were not suitable for embedded computer systems. An *embedded computer* is one that is a component of a larger system, e.g., a guidance computer on a missile, or a target tracking computer in a weapon system, etc. [Ref. 4]

The United States Department of Defense (DoD) recognized this software crisis. There was no standardization in DoD projects as far as programming languages were concerned. In response to these problems, DoD commissioned a worldwide language design competition with the goal of developing a single high-order language suitable for use in embedded computer systems as well as conforming to other well-defined requirements. The result of that competition was Ada. It is designed specifically for large, software-intensive, real-time embedded computer systems. However, it is suitable for other application areas as well. Ada both embodies and enforces modern software engineering development principles, such as structured constructs, information hiding,

and abstraction. Ada is now a prime language in the computing industry, as well as being the required language for embedded computer systems in DoD projects. [Ref. 4]

Both viewpoints on the definition of a fourth-generation language are applicable to this thesis. In the first sense, that of an application generator, the Synthesizer Generator is a CASE tool that takes as input an editor specification written in a fourth-generation language and outputs compilable C code, which is then compiled into an executable editor program. In the second sense, that of a class of programming languages, the purpose of the editor produced by the Synthesizer Generator is to support the early stages of software design for eventual translation in to a fourth-generation language, specifically Ada.

II. BACKGROUND

A. GENERAL DESCRIPTION OF A SYNTAX DIRECTED EDITOR

A syntax directed editor is a software tool for creating or modifying programs in such a way that correct syntax is always maintained. It will not allow syntactically incorrect constructs to be entered. Generally, the editor is designed for one specific programming language, although editors exist that allow different languages to be programmed into it. Often, the editor is the prime component of a set of programming tools that make up an integrated programming environment.

Most syntax directed editors are screen-based, i.e., program text is entered at the location indicated by the display cursor, and the screen display is automatically updated, just as most text editors do. But text editors, as the name implies, only allow modification of the *text*, whereas syntax directed editors allow modification of the *syntactic structure* of the program, as well as the text itself. As the text is entered, it is checked for syntactic correctness, and errors are immediately detected and appropriate messages displayed to the user. This prevents the user from writing programs that are syntactically incorrect. [Ref. 5,6]

In most syntax directed editors, text is entered by creating a template, or a skeleton, of some syntactic construct, and filling in the detail later. These templates form a parse tree and only allow syntactically correct constructs to be generated. The user selects a node of the parse tree that contains a non-terminal to fill in the detail. This non-terminal

can be expanded into valid syntactic alternatives. Some editors will display a list a valid choices for the current node. The user can also enter directly the desired text, and if the entry is not syntactically correct, the editor will then display a list of alternatives to the user. This process, known as *programming by selection*, is particularly beneficial in a learning environment because the time to learn the rules of the language (the syntax) is reduced. Figure 2 is an example of the programming by selection process for a compound statement using an arbitrary language. [Ref. 5,6]

<u>Program Statement</u>	<u>Transformation Selected</u>
begin <stmtlst> end	
begin <stmtlst>; <stmt> end	<stmtlst> $\longrightarrow$ <stmtlst>; <stmt>
begin <stmt>; <stmt> end	<stmtlst> $\longrightarrow$ <stmt>
begin <assign_stmt>; <stmt> end	<stmt> $\longrightarrow$ <assign_stmt>
begin x := 5; <stmt> end	<assign_stmt> $\longrightarrow$ x := 5
begin x := 5; <proc_call> end	<stmt> $\longrightarrow$ <proc_call>
begin x := 5; p(x) end	<proc_call> $\longrightarrow$ p(x)

Figure 2
Programming by Selection

Since visual display units (VDUs) generally can only display 24 lines of text at a time, it is impossible to display much of the parse tree graphically on the screen. However, an overall view of the program (or at least a larger portion of it) is possible by *unparsing* the tree, i.e., displaying text down to a certain level of detail and omitting the

remaining detail. The omitted portions are commonly replaced by ellipses ("...") so the overall view of the structure remains intact. The user may then select a particular portion on which to zoom in and display greater detail. [Ref. 5]

Changes are made to the structure of the program using the central commands of the editor. The editor might allow changes in terms of operations on the branches of the parse tree, by matching subtrees and substitution, or by entering arbitrary text and reparsing the tree. For example, suppose the user wanted to rename a variable "red" to "blue". Variables are a construct of the language, i.e., a subtree. Changing the symbol of this subtree from "red" to "blue" does not affect other text strings in the parse tree that contain the substring "red", i.e., a variable "fred" is not changed to the variable "fblue". This kind of structural editing (as opposed to simple text editing) avoids unintended changes to the program. If this were not the case, i.e., if such a substitution replaced *every* instance of the substring "red" throughout the program, the result would be potentially disastrous. [Ref. 5]

Structural commands insert, copy, move and delete subtrees within the parse tree. Subtrees can also be *clipped*, or contracted into a non-terminal symbol of the grammar. These clipped pieces are saved, and can be reused at other locations using a special form of the insert command. The benefit of this facility is the time saved by copying or moving whole subtrees from one location in the parse tree to another rather than having to re-construct the subtree at the new location. [Ref. 5]

Most syntax directed editors incorporate a pretty-printing function. This can be reflected on the display screen as well as on the hardcopy output. Pretty-printing might be a selectable function, or it might be done automatically. [Ref. 5]

A syntax directed editor can also maintain some semantic constraints. For example, every variable that is used must be declared. Such facilities are useful for preventing some classes of errors.

The SPECDEF editor incorporates some of the facilities mentioned above. It is a screen-oriented hybrid editor that modifies the structure of the edited object by template insertion as well as direct text entry. The valid choices for template insertion are displayed to the user at each node of the parse tree when that node is selected. Textual changes to one construct do not affect the same text in other constructs. Subtrees can be clipped and copied or moved to other locations in the parse tree. Pretty-printing is automatic in the sense that indentation and line breaks are designed into the unparsing schemes.

SPECDEF does not, however, have any provisions for semantic checking of variable declarations or for elision of detail into ellipses. Future versions of SPECDEF will be integrated with a type checker and a consistency checker that will help maintain semantic constraints. Provisions for elision of detail may also be incorporated in future versions of the SPECDEF editor.

B. SYNTAX DIRECTED EDITOR GENERATORS

An important area of research in computer science is that of *language-based programming environments*. Probably the most significant programming tool in such an

environment is a syntax directed editor. Integrating a syntax directed editor with execution and debugging facilities creates a powerful programming tool. Several such systems have been developed, e.g., GANDALF [Ref. 7,8], and the Cornell Program Synthesizer [Ref. 9]. The Synthesizer Generator [Ref. 10,11] is an outgrowth of the Cornell Program Synthesizer.

Both GANDALF and the Synthesizer Generator have the capability to create editors for languages other than programming languages, such as verification languages or specification languages. The Synthesizer Generator was selected as the primary tool for this thesis because of the availability of mutual support with a parallel research project in prototyping languages currently under development at the Naval Postgraduate School. Also, at this stage of research, there was not a need to generate a multiple user environment such as GANDALF generates. The discussion on GANDALF is offered for purposes of comparison. Follow-on research to this thesis may wish to investigate the use of GANDALF to create a multiple user environment.

1. The GANDALF System

The GANDALF System is a software tool that generates integrated environments that are language-specific. Two different kinds of environments are possible using this system--*programming environments* and *software development environments*. A programming environment is a highly interactive knowledge-based environment for a single programmer working on a small project. A software development environment, which is the primary purpose of the GANDALF project, is one in which multiple programmers work on large projects. [Ref. 8]

The heart of the GANDALF project is designed around the ALOE editor generator system, which stands for A Language Oriented Editor. A language oriented editor is also known as a syntax directed editor or a structure editor in the literature. ALOE is actually the common kernel that is used by all syntax directed editors generated with this system. A specific instantiation of a syntax directed editor is known as an ALOE *editor*. ALOE GEN is an ALOE editor that is used to create the syntactic descriptions of a language that can then be used in conjunction with ALOE LIB, which is library support to add semantic actions and additional commands, to create a syntax directed editor for the specific language. [Ref. 7]

The user interface of all generated ALOE editors are tree-oriented full-screen interfaces. There is a set of language-independent commands, such as subtree deletion and cursor motion, as well as a set of language-specific operations that represent *structures of the language*, such as Pascal **while** statements or Ada **packages**. [Ref. 7]

As a programmer creates or edits a program with an ALOE, the program is displayed much the same way as a text editor would display it. But the representation is based on the *structure* of the program rather than simply as a string of characters. This structure is an abstract syntax tree. The programmer moves about the program by way of this tree structure. The cursor will highlight an entire structural unit, such as a Pascal **if-then-else** construct, or a specific non-terminal within the structural unit, rather than just a single character. Operations are performed on the structure of the program, i.e., insertion or deletion of structural units, or expansion of non-terminals. [Ref. 8]

As stated earlier, the primary focus of the GANDALF project is the generation of software development environments that would support multiple programmers working on a large project. This type of environment also has an ALOE editor as its basis, but in addition provides a system version control editor and a project management editor to solve programming-in-the-large and programming-in-the-many problems respectively. These problems are significantly harder to solve because of the interaction required between various modules developed by different programmers on the team. Research is continuing in this area. [Ref. 8]

2. The Synthesizer Generator

The Synthesizer Generator is a system that creates a language-specific editor from a specification of the language's abstract syntax, context-sensitive relationships, display format, concrete input syntax, and transformation rules for restructuring objects. The specification language for the Synthesizer Generator, SSL (for Synthesizer Specification Language), is based on the concept of an *attribute grammar*. An attribute grammar is an extension of a context-free grammar. The extensions are the *attributes* which have been attached to the nonterminal symbols. The value of these attributes is defined by *attribute equations*. This mechanism can specify how widely separated parts of a derivation tree can be constrained by the context of the rest of the tree. [Ref. 10]

Specifications written in SSL are different from specifications written for other systems that are based on attribute grammars. This is because SSL has several innovations, e.g., the capability to merge abstract-syntax definitions and user-defined

attributes, a notation that allows specifications to be broken into separate modules, and the manner in which the parser is incorporated into the system. [Ref. 10]

There are two basic components that comprise the Synthesizer Generator: (1) a translator that converts SSL specifications into various tables as output, and (2) an editor kernel with an attributed-tree data type and a driver for manipulating attributed trees interactively. The kernel executes operations on the current tree according to keyboard and/or mouse commands. A shell program named **sgen** invokes the translator and produces the syntax directed editor for the specified language from the tables that are output from the translator. [Ref. 11]

Objects being edited with an editor produced by the Synthesizer Generator are represented as attributed derivation trees. Each modification to the object being edited causes the attribute values throughout the tree to be updated. If a modification results in the violation of any context-dependent constraints, the display is annotated with error messages to provide the user with immediate feedback. [Ref. 11]

The editor produced by the Synthesizer Generator displays constructs of the language as *templates*, or predefined, formatted *patterns*. Non-terminals within constructs can be thought of as *placeholders* where additional insertions can be made through a *transformation* equation. When the user selects a subterm (i.e., a placeholder) a menu of possible transformations is listed. A transformation is invoked either by typing the name or making a menu selection with a mouse. Transformation definitions are type-checked when the SSL specification is compiled into an editor, so it is impossible for a transformation to introduce context-free syntax errors. [Ref. 11]

In addition to the transformations, which are language specific, every editor generated by the Synthesizer Generator contains language independent *system commands* such as **cut-to-clipped** and **paste-from-clipped**. The cut operation differs from ordinary text editors, which would delete the selected text and leave the cursor at that position. In the syntax directed editor, the cut operation replaces the selected subterm with a placeholder, thus maintaining the correctness of the context-free syntax. The paste operation, conversely, replaces the placeholder term with the previously cut subterm.[Ref. 11]

The Synthesizer Generator is written in C and runs under the Berkeley UNIX system. Editors can be generated for specific windowing systems, such as VIDEO, SUN, and X-Windows. There are a number of selectable options when invoking **sgen** that affect the method of generation and the capabilities of the generated editor. For a complete list of these options, refer to Appendix B of [Ref. 11].

III. AN OVERVIEW OF THE SPEC SPECIFICATION LANGUAGE

A. THE PURPOSE OF SPEC

The functional specification stage of the software development cycle gives a black-box specification of the software project, i.e., only the interface between modules is described, not the inner workings. Spec is a formal language for writing such black-box specifications, as well as for specifying the internal interfaces of the proposed system during the architectural design phase. [Ref. 12]

A precise, formal specification of a proposed software system is essential in order to ensure that everyone on the development team understands and agrees with the interpretation of the user's requirements. Programming languages, such as Ada, are formal, but because they are designed for describing algorithms and data structures rather than the actual behavior of a module, they are not suited for writing black-box specifications. Spec is designed specifically for defining the behavior of software modules. The formal notation of Spec provides the precision needed to prevent the ambiguity inherent in English and other informal notations. [Ref. 12]

Spec uses the event model to define the behavior of a proposed system. The event model describes computations in terms of modules, events and messages.

A module is a black box that interacts with other modules only by sending and receiving messages. An event occurs when a message is received by a module at a particular instant of time. A message is a data packet that is sent from one module to another. (Berzins, 1987, p. 2)

Modules can model either software components, or external systems such as peripheral hardware or even the user. Messages are accepted by a module one at a time. The length of a message is arbitrary. It is assumed that every message sent eventually arrives at its destination. [Ref. 1]

The sequence of messages that are received by a module determine what kind of response will occur. Modules are classed as **mutable** or **immutable** depending on how the module reacts to messages. If the response of the module can depend on one or more messages received prior to the most recently received message, the module is mutable. Mutable modules behave as if they had internal states or memory. If the module's response to every possible message depends solely on the most recently received message, the module is immutable. A module is immutable if and only if it is not mutable. Immutable modules behave more like mathematical functions. [Ref. 12]

B. MODULE DESCRIPTIONS

Five types of modules can be specified in Spec: definitions, functions, state machines, abstract data types, and instances of *generic modules*.

1. Definition Modules

Definition modules contain descriptions of concepts that are not unique to any particular function, machine, or type declaration, but are instead available to be shared among many modules through importation. Only concepts may be declared in definition modules. *Definition modules may also import other modules.* [Ref. 1]

Concepts describe the logical assertions that define the behavior of modules. They can define constant symbols, symbolic type names, predicates (relationships) and

functions (attributes). These conceptual representations are introduced by the keyword **CONCEPT**. Concepts can be explicitly exported to other modules through an **EXPORT** clause, as well as imported from other modules through an **IMPORT** clause (provided it has been explicitly exported from the module in which it is defined). [Ref. 1]

Definition modules form the primary subset of the requirements analysis phase of the software development cycle. In requirements analysis, a model of the problem domain is created to record facts about the problem and the environment of the proposed system needed by the developers in later stages. These facts are represented as concepts contained in definition modules. The SPECDEF tool developed in this thesis treats the subset of the Spec language used in this process.

2. Function Modules

Function modules are one of the three primary module types in Spec (the others being machine modules and type modules). A function module behaves like a mathematical function in that it calculate the value on a data type. [Ref. 12]

A function usually will perform only one task or service, so they will only accept anonymous (i.e., unnamed) messages. Messages in a function module define the "operations" that the function may perform. They are introduced by the keyword **MESSAGE**. [Ref. 1,12,13]

Each message, or operation, returns a value in the form of a **REPLY** message. A variety of responses can be defined with **WHEN** clauses, which specify preconditions similar to the Pascal **case** statement or the **switch** statement in C. If none of the **WHEN** clauses apply, an **OTHERWISE** clause handles the remaining cases. If a message

response is to report an abnormal condition, a REPLY EXCEPTION message is returned. Postconditions that must be satisfied by the outgoing message are specified with a WHERE clause. [Ref. 1,12,13]

To illustrate the above concepts, Figure 3 depicts a function *f* that accepts an anonymous message consisting of an input variable of type *type1*. If the input variable satisfies precondition *case1*, the returned value is the output variable of *type2*, with a *postcondition* that must also be satisfied. If the input variable satisfies precondition *case2*, an abnormal situation exists, so the returned value is the exception *failure1*. If neither of the preconditions can be met, then the OTHERWISE clause takes effect and an exception *failure2* is returned.

```
FUNCTION f
  MESSAGE (input-var : type1)
    WHEN case1 (input-var)
      REPLY (output-var : type2)
      WHERE postcondition (output-var, input-var)
    WHEN case2 (input-var)
      REPLY EXCEPTION failure1
    OTHERWISE REPLY EXCEPTION failure2
END
```

Figure 3
Example of a Function Module

3. Machine Modules

Machines are mutable modules that have an internal state. The conceptual model of that state describes the behavior of the module, rather than the messages received in the past. The keyword STATE introduces the declaration of the components of the conceptual model of the state, and any restrictions on the set of meaningful states are

given in an INVARIANT clause. Restrictions on the initial state are given in an INITIALLY clause. Input stimuli to the machine which invoke actions are defined in MESSAGE clauses. The actions can be any combination of a REPLY, a REPLY EXCEPTION, one or more SEND statements, or a TRANSITION to a new state. Transitions are given as equations that describe the change either forwards or backwards in time, whichever is simpler (i.e., in terms of the state before the transition or in terms of the state after the transition). Messages can be sent to modules other than the origin of the incoming message. This is done using a SEND statement instead of a REPLY. There can be any number of SEND statements, but only one REPLY statement. [Ref. 1,12,13]

The example in Figure 4 depicts a machine called *buffer* which has a state model consisting of a single-valued *state-var* of type *type1*. The statements INVARIANT *true* and INITIALLY *true* indicate there are no restrictions on the states of the machine. If the stimulus *read* is sent to the machine, the reply is the current value of *state-var*. If the stimulus *update* is sent with the parameter *new-value* of the type *type1*, the state of the machine is changed by assigning the *new-value* to *state-var*. When the transition is complete, the reply message *done* is sent to the initiator of the stimulus.

```

MACHINE buffer
  STATE (state-var : type1)
  INVARIANT true
  INITIALLY true

  MESSAGE read
    REPLY (result : type1)
    WHERE result = state-var

  MESSAGE update (new-value : type1)
    TRANSITION state-var = new-value
    REPLY done

END

```

Figure 4
Example of a Machine Module

4. Type Modules

Abstract data types are defined in type modules. An abstract data type consists of a set of values and a set of primitive operations that operate on those values. In the type module each operation is represented by a named message. The data type is described in terms of a conceptual model rather than the actual data structure used in the implementation. The implementation of the data type can change to improve performance, but the conceptual model will still be valid. The keyword **MODEL** introduces the tuple that represents each instance of the data type. The components of the tuple can have restrictions specified in an **INVARIANT** clause. The effects of the operations can be described by **CONCEPT** clauses. Type modules can be either mutable or immutable. [Ref. 12]

5. Instance Modules

An instance module is used to make an instance or partial instantiation of a generic module. This is useful for renaming concepts in other modules before importing or inheriting them. For example, if module *m* contains a concept *c* that you wish to import, but you have already defined a local concept using the same name and argument types but with a different meaning, one of the two concepts will have to be renamed. If the name you have chosen for your local concept is a good one, then it is better to rename the imported concept. To do this, you must first create a new instance of module *m*, say *new-m*, in which you declare a renaming of concept *c* to *new-c*. The renamed concept can now be imported from the new instance of module *m* (i.e., module *new-m*). This example is illustrated in Figure 5.

```
INSTANCE new-m = m
    RENAME c AS new-c
END

-- The renamed concept can now be imported as follows:
IMPORT new-c FROM new-m
```

Figure 5
Using an Instance Module to rename a concept

C. OTHER FEATURES

1. Concepts used as Subprograms

Concepts, which were briefly described above, are of great benefit in simplifying descriptions of complex software systems. Long expressions in predicate logic, for

example, can be decomposed into several concepts that are easily understood individually, making the overall expression more understandable. Concepts in Spec are analogous to subprograms in a programming language.

2. Inheritance in Spec

Spec has an inheritance mechanism that allows constraints that are common to the interface of several modules to be specified. This is particularly useful in specifying large software systems because it helps achieve a consistency in the interface of several modules. [Ref. 12]

Inheritance applies to entire modules. An inherit clause is introduced with the keyword INHERIT followed by the name of the module being inherited. The effect is the same as if all the concepts, messages, models or states of the inherited module were copied verbatim into the current module. Any specific concept, message, model or state may be excluded from being inherited into the current module through a HIDE clause. To avoid name conflicts, messages or concepts can be "renamed" to a "new" name with a RENAME clause. [Ref. 13]

There are restrictions on which type of modules can be inherited by which other types of modules. Definition modules may only inherit other definition modules. Function modules may inherit other functions and definition modules. Machine modules may inherit other machines, function modules and definition modules. Type modules may inherit other types, function modules and definition modules. Any of these four type of modules may inherit an instance module provided the base module that the instance instantiates is of a type appropriate to the module that is doing the inheriting. [Ref. 13]

A more detailed discussion of inheritance in Spec can be found in [Ref. 13].

Closely related to inheritance is the mechanism for importing and exporting concepts from one module to another. Only CONCEPTS may be imported and exported. An IMPORT clause is used to allow concepts defined in other modules to be made a "part of" the current module. An EXPORT clause allows concepts defined in the current module to be available for explicit importation into other modules. A concept cannot be imported unless it has been explicitly exported by the module in which it is defined. This importation and exportation mechanism, as well as the inheritance mechanism, is useful for logical grouping of related concepts into a single module in support of a modular construction of large software systems. [Ref. 1,12,13]

The SPECDEF editor is able to declare INHERIT, IMPORT, and EXPORT clauses for definition modules. However, there is no capability for checking that the named module in an INHERIT clause exists, or whether a concept named in an IMPORT clause has been EXPORTed by the named module. It is therefore up to the user to ensure that such is the case.

D. COMPLETE DESCRIPTION OF SPEC

The foregoing discussion on Spec is necessarily brief. For a more thorough description see [Ref. 12] and Chapter 3 of [Ref. 1]. The grammar of the Spec language is included at the end of this thesis as Appendix A.

IV. DESIGN OF THE SYNTAX DIRECTED EDITOR FOR SPEC

A. GOALS

1. Short Term

The initial short term goals for this research project were as follows: (1) gain an understanding of how to write a specification for a syntax directed editor in terms of the Synthesizer Generator specification language, SSL, and (2) implement a working model of the syntax directed editor for a subset of Spec. In order to achieve these goals as quickly as possible, it was decided to limit the specification to syntax editing only and forego any semantic checking.

2. Long Term

The long term goals (beyond the scope of this thesis) are to (1) implement the editor for the complete Spec grammar, and (2) integrate the editor into a complete programming environment for Spec. This environment will include the editor, a type checker [Ref. 15], a pretty printer [Ref. 16], a consistency checker, an inheritance expander, a test oracle, a diagram generator, and translator to Ada. With the exception of the pretty printer and an initial version of both the editor and the type checker, the tools mentioned above have yet to be developed.

B. CONCEPTS FOR USING THE SYNTHESIZER GENERATOR

To effectively utilize the Synthesizer Generator, several terms and concepts that are perhaps unfamiliar to most readers must be understood. An SSL editor specification is

built around three different but related grammars: the *abstract syntax* grammar, the *concrete syntax* grammar, and the *unparsing scheme*. The abstract syntax defines the basic set of context-free grammar rules that define the language, excluding any keywords in context (hence, context-free). The concrete syntax defines the input grammar for text entry, and the productions of this grammar *do* take keywords into account. The unparsing scheme defines the formatting rules for the display representation of the language, including keywords, punctuation, and indentation. [Ref. 10,11]

1. Attribute Grammar

The Synthesizer Generator is based on the concept of an *attribute grammar*. As mentioned previously, an attribute grammar is a context-free grammar that is extended by attaching *attributes* to the nonterminal symbols of the grammar and by defining *attribute equations* that determine the value of those attributes. Each *attribute* is a piece of information that describes some semantic property of the nonterminal. For example, a variable could have attributes such as *type*, *value*, *length*, etc. Each production in the grammar can have a set of attribute equations.

There are two disjoint sets of attributes of a nonterminal: *synthesized* attributes and *inherited* attributes. Synthesized attributes are associated with the left-hand-side nonterminal of a production, while inherited attributes are associated with nonterminals of the right-hand-side of a production. That is, the left-hand-side *synthesizes* its attribute values from the right-hand-side attributes, and the right-hand-side *inherits* its attribute values from the left-hand-side attributes. [Ref. 10: p. 39]

The current version of the SPECDEF editor defines only a synthesized attribute t for some nonterminals whose value is the text string entered to represent the nonterminal. Future versions of SPECDEF may incorporate attributes that pertain to error messages or other environmental factors related to type checking.

2. Abstract Syntax, Phyla and Terms

The underlying *abstract syntax* of the language is simply the set of grammar rules that define the language, excluding any keywords. In SSL, the abstract syntax consists of a set of productions of the form

$$X_0 : op (X_1 X_2 \dots X_k);$$

where *op* is an *operator* name and each X_i is a nonterminal of the grammar. Each of these nonterminals is the name of a *phylum*, and each phylum is the set of derivation trees that can be derived from the nonterminal. A phylum is a special kind of abstract data type. The derivation trees are known as *terms*. A term is an expression that applies a k -ary operator to k elements of an appropriate phyla. The SSL grammar rule acts the same as the context-free production:

$$X_0 \longrightarrow X_1 X_2 \dots X_k$$

The operator serves to distinguish between rules that have right-hand-sides that are structurally identical but which may have different keywords in a concrete syntax. For example, the operator names Equal, Add, and Subtract in Figure 6 distinguish between three kinds of expression pairs. [Ref. 10: pp. 45-47]

root start;		
start	:	Spec(spec);
optional list spec;		
spec	:	ModuleNil()
		ModulePair(module spec)
	:	
module	:	DefModuleDecl(interface concepts);
interface	:	InterfaceDecl(formal_name inherits imports export);
...		
where	:	WhereEmpty()
		WhereExp(expression_list)
	:	
optional list concepts;		
concepts	:	ConceptNil()
		ConceptPair(concept concepts)
	:	
concept	:	ConceptType(formal_name type_spec where)
		ConceptValue(formal_name formal_arguments where
		formal_arguments where)
	:	
formal_name	:	FormName(identifier formal_parameters);
...		
expression_list	:	SingleExp(expression)
		MultiExp(expression_list expression)
	:	
expression	:	UndefExp()
		Equal(expression expression)
		Add(expression expression)
		Subtract(expression expression)
	:	
identifier	:	Identifier(IDENTIFIER);
IDENTIFIER	:	IdentLex< [a-zA-Z][a-zA-Z_0-9]* >;

Figure 6
Partial Listing of Abstract Syntax for Spec

The abstract syntax of a language must have one phylum designated as the *root phylum*. The objects that can be edited within the generated editor are terms of this root phylum. [Ref. 10: p. 48]

3. Completing Term

Each phylum has a default value called the *completing term*. The editor uses these default values at each unexpanded occurrence of a phylum in the derivation tree of the object being edited. [Ref. 10: p. 48]

The completing term of a phylum is constructed by the first operator declared for that phylum, such as the operator `UndefExp` of phylum `expression` and the operator `WhereEmpty` of phylum `where` in Figure 6. This *completing operator* is applied to the completing terms of its argument phyla. (List phyla or optional phyla are exceptions to this rule, as described in Sections IV.B.5 and IV.B.6 below.) [Ref. 10: pp.48] For example, the completing term for phylum `module` defined in Figure 6 is the term

```
DefModuleDecl(InterfaceDecl(FormName(...),InheritNil,ImportNil,ExportNil),ConceptNil)
```

i.e., the completing operator for `module` applied to the completing terms of phyla *interface* and *concepts*. Notice that the completing term for phylum `interface` had to be constructed in the same manner before the completing operator of phylum `module` could be applied to it.

4. Placeholder Term

Each phylum has an associated *placeholder term* which is also a default representation for its respective phylum. The placeholder is used to represent an "unexpanded" node in the derivation tree of the object being edited, i.e., it represents a "to be determined" value for the respective phylum. [Ref. 10: p. 69]

Phyla declarations may have *property declarations* that determines the behavior of the placeholder term. The property declarations can declare a phyla as an *optional*

phylum, a *list* phylum, or an *optional list* phylum. Any phylum that does not have one of these declared properties is referred to as an *ordinary* phylum. For ordinary phyla and non-optional list phyla, the placeholder term and the completing term for the respective phylum are identical. However, for phyla declared as optional or an optional list, the concepts are different as explained in Sections IV.B.5 and IV.B.6. [Ref. 10: p. 77]

The placeholder term often displays the default representation of its phylum as a string of characters consisting of the phylum name enclosed in angle brackets, e.g., <concept> or <identifier>. Such a display representation is not mandatory, however. For example, in the SPECDEF editor, the placeholder term (and completing term) for each of the phyla field, type_spec, and expression is the symbol ?, which is the Spec notation for an undetermined value that must be defined later.

In any case, the placeholder term is used at an unexpanded node in the derivation tree. Even though the derivation tree contains placeholders, it is still a complete derivation tree from the system's point of view. [Ref. 10: p. 49]

5. Lists and Optional Lists

Certain phyla are designated as *lists*. A list represents a sequence of items, such as an argument list in a subprogram. Phyla declared as lists must have exactly two operators, one being a nullary operator and the other a binary operator that is right recursive. In the example shown in Figure 7, phylum stmtList is declared as a list phylum, with the nullary operator StmtListNil and the binary, right recursive operator StmtListPair. [Ref. 10: p. 49]

list stmtList;	
stmtList	:
	StmtListNil()
	StmtListPair(stmt stmtList)
	;

Figure 7
Declaring a list phylum

List phyla can be declared as *optional lists*. Declaring a list as optional causes it to behave as a list of zero or more elements, whereas a non-optional list behaves as a list with at least one element. [Ref. 10: p. 82]

The completing term for list phyla and optional list phyla is defined differently than for ordinary phyla. for a non-optional list, the completing term is constructed by applying the binary operator to the completing term of the left-argument phylum and to the list's nullary operator. This has the effect of concatenating the completing term of the left-argument phylum with the nullary operator of the list phylum, resulting in a singleton list. For optional list phyla, the completing term is simply the constant term formed from the nullary operator of the phylum. [Ref. 10: pp. 77-82]

The placeholder term for list phyla (whether optional or non-optional) is formed the same as the completing term had the phylum been declared as a non-optional list. As an example, the completing term for phylum `spec` in Figure 6 is the term

`ModulePair(DefModuleDecl(InterfaceDecl(...), ConceptNil), ModuleNil)`

i.e., `ModulePair` applied to the completing term of phylum `module` and to the term `ModuleNil`. [Ref. 10: pp. 79-82]

Editors generated by the Synthesizer Generator provide special built-in actions for manipulating list phyla. The system command **forward-with-optionals** <^M or RETURN> or **forward-preorder** <^N> is used to move the selection point forward in a derivation tree to the next node in a preorder traversal (see Appendix C for information on executing system commands). When moving the selection through a list, the editor automatically inserts a placeholder term before and after each list element. If the placeholder term is not edited (i.e., transformed through template insertion or text entry, both described later), when the selection is moved to an element that is not contained within the placeholder term, the placeholder is deleted from the derivation tree. Because this insertion and deletion process is automatic, these placeholder terms are referred to as *transient placeholders*. [Ref. 10: pp. 69-72]

The qualifier **optional** introduces a distinction between a phylum's completing term and its placeholder term. As stated previously, for ordinary phyla and non-optional list phyla, the completing term and placeholder term of the respective phylum are identical. When the current selection is an optional list phylum, the completing term is automatically replaced by the phylum's placeholder term. If the placeholder term is not edited through template transformation or text entry (both described later), and the selection is moved outside the scope of the placeholder, the placeholder is automatically replaced by the completing term. [Ref. 10: pp. 76-82]

6. Optional (Non-List) Phyla

Elements of the language that are optional can be declared as optional phyla. Optional list were discussed in the previous section. Non-list phyla can also be declared as optional. [Ref. 10: pp. 77-82]

An optional (non-list) phylum can have any number of operators, but one must be a nullary operator. The completing term of the optional phylum is formed from the first listed nullary operator of the phylum in the editor specification. The placeholder term is formed by "completing" the first operator of the phylum that is not the completing-term operator. Since the completing term is formed by a nullary operator, it will contribute nothing to the display representation. Therefore, the order of the operator declarations should be arranged such that the placeholder term displays the appropriate prompt, e.g., the phylum name enclosed in angle brackets or some other appropriate display string. [Ref. 10: pp. 79-80]

7. Lexical Phyla

A *lexical phylum* is used to declare special strings of characters called *lexemes* that represent the smallest lexical units of the language, such as keywords, punctuation and other special characters, or literal constants such as identifiers, integers, etc. The three grammars that are used in an editor specification are interrelated, and lexemes play an important role in these relationships. In the relationship between the abstract and concrete grammars, lexemes are used to differentiate between similar constructs during semantic analysis. It should be noted that in the concrete grammar, certain lexemes can have more than one version or style. In the unparsing scheme, lexemes are used to

display keywords, punctuation, other special characters such as operator symbols, or constants. [Ref. 11: p. 15]

Lexemes are defined by one or more regular expressions that are recognized by the lexical analyzer generator Lex [Ref. 17]. The form of a *lexeme declaration* is

phylum-name : *lexeme-name* < *regular-expression* >;

which declares that all strings generated by the given *regular-expression* are in the named phylum. There must be at least one blank separating the regular expression from the closing angle bracket. The regular expression itself must contain no embedded blank characters except those explicitly escaped by a preceding backslash. The *lexeme-name* is used in the definition of the concrete input grammar to represent an instance of the actual lexeme, such as a keyword or special symbol. [Ref. 11: pp. 15-17]

The regular expressions in a lexeme declaration are exactly the regular expressions accepted by Lex, with only a few exceptions as follows [Ref. 11: p. 16]:

- The blank character within square brackets must be escaped.
- Definitions, as described in Section 6 of the Lex manual [Ref. 17], are not supported.

In Figure 8, "C" stands for any printable character, "N" stands for any decimal integer, and "E" stands for any regular expression. Each listing is a regular expression. The symbols " \ [] - ^ . \$? * + | () / { } % < > " each have special meaning in regular expressions (see [Ref. 17] for full details). If they are used as literal text characters, they must appear within quotation marks or be escaped with a preceding backslash. Inside square brackets, however, only the characters \ - ^ and blank have special meaning. All other characters denote themselves. [Ref. 11: pp. 16-17]

<u>Expression</u>	<u>Meaning</u>
C	the character C
$"C_1C_2C_3"$	the string $C_1C_2C_3$
$\backslash C$	the character C
$[C_1C_2C_3]$	the character C_1 , C_2 , or C_3
$[C_1-C_3]$	any of the characters from C_1 through C_3
$[^C_1C_2C_3]$	any character but C_1 , C_2 , and C_3
$\cdot$	any character but newline
^{E}	an E at the beginning of a line
$E\$$	an E at the end of a line
$E?$	an optional E
E^*	0 or more instances of E
E^+	1 or more instances of E
E_1E_2	an E_1 followed by an E_2
$E_1 E_2$	an E_1 or an E_2
(E)	an E
E_1/E_2	an E_1 but only if followed by an E_2
$E\{N_1N_2\}$	N_1 through N_2 occurrences of E

Figure 8
Construction of Regular Expressions

Lexeme declarations are not totally independent, since their order can influence the recognition process during lexical analysis. When more than one regular expression can be matched, the longest match is made. If several rules match the same number of characters, the declaration defined earlier in the specification is used. Because of this, all keywords should be defined prior to a definition for a class of identifiers. [Ref. 11: pp. 16-17]

Whitespace characters (spaces, tabs, newlines) are ignored during parsing, but are important for the concrete input syntax. The lexeme declaration

WHITESPACE: $< [\backslash \backslash n \backslash t] >$;

defines such characters for the system. [Ref. 11: p. 17]

C. PRACTICAL APPROACH TO DESIGNING AN EDITOR

There are many aspects of an editor specification. A novice editor-designer will probably be confused on where to begin. The following discussion is the recommended approach by the authors of the Synthesizer Generator together with insights gained through practical hands-on experience.

Start by choosing a simple subset of the full language. This allows the editor-designer to become familiar with the formats of the various sections of the editor specification without getting lost in the details required for the full language. Once the editor is implemented for this subset, and works as expected, the subset can be expanded. The remaining steps are listed in Table 1 below and explained in more detail in the rest of this section. [Ref. 10: pp. 162-169]

Table 1: STEPS TO DESIGNING AN EDITOR SPECIFICATION

1. Define the abstract syntax.
2. Define the display representation.
3. Define template transformations.
4. Generate and debug editor.
5. Define concrete input syntax.
6. Refine the display representation.

It is recommended to design the editor specification in a modular fashion. This enhances the understandability of the editor specification. It also supports the reusability of the different "modules" in related editors, such as upward-compatible editors for language extensions or different editors for the same language that provide different

display formats. Reps and Teitelbaum recommend [Ref. 10: pp. 172-178] the following six modules for organizing an editor specification:

- Abstract-syntax declarations and template transformations on these constructs.
- Lexical declarations.
- Concrete-input syntax declarations together with precedence and associativity rules.
- Attribute-domain declarations and operations on the attribute domain. (Note: the current version of SPECDEF does not have any attribute-domain declarations, such as environment attributes. However, future versions may need a file with these kinds of attributes.)
- Attribute declarations and equations.
- Unparsing declarations.

It should be noted that the above order of modules does not correspond to the order of steps for designing an editor specification. This is the recommended order that the respective SSL files should be input to the Synthesizer Generator to create an editor.

1. Define the Abstract Syntax

The first step is to define the abstract syntax of the language subset. A Backus-Naur Form or context-free grammar is a convenient starting point. Minimize unnecessary syntactic distinctions. For example, in the production

`exp :     Const(INT);`

the numerals 007 and 7 will both be translated to the INT value 7, and displayed as the numeral 7. If the distinction between 007 and 7 must to be preserved, however, the pre-defined phylum INT should be replaced by the phylum, say, INTEGER with the lexeme

declaration

INTEGER: < [0-9]+ >;

which defines numerals as strings. Leading zeros are thus preserved. [Ref. 10: p. 163]

Omit all terminal symbols that are just "syntactic sugar", such as punctuation marks and keywords. Operator names will suffice to distinguish between alternative terms of the left-hand-side phylum. Only those terminals that carry semantic information should be retained in the abstract syntax, such as identifiers, numerals, and other literal constants. [Ref. 10: pp. 163]

Attribution schemes can influence the design of a language's abstract syntax. Phrases that are lexically identical are often used for distinct purposes. The static-semantic analysis then depends on the context in which the phrase is used. If all usages of the phrase have a common syntax, then the context must be passed down as an inherited attribute to select the appropriate analysis. However, if each use of the phrase occurs in a different phylum, the context is implicit in the phylum's operators, and the correct attribute equations for the static-semantic analysis are automatically selected. It is an unavoidable fact that the abstract syntax may have to be changed when attribution rules are addressed. [Ref. 10: p. 164]

2. Define a Display Representation

The second step in specifying an editor is to define an initial set of unparsing declarations to allow the editor to display the terms of the abstract syntax. Do not be concerned with fancy pretty-printing at this point. Define only enough "syntactic sugar" to debug the abstract syntax. Specifically, do not consider alternative unparsing schemes

(an alternate display representation of the given production, often used for an "abbreviated" display), optional line breaks, context-dependent display formats, and special fonts now. However, line breaks and simple indentation rules are advisable at this stage. [Ref. 10: p. 164]

The unparsing declarations define which productions of an object are editable as text as well as the display format of a term. There is an unparsing production corresponding to each production of the abstract syntax. Each unparsing production consists of a sequence of strings, names of attribute occurrences, names of right-hand-side phylum occurrences, and *selection symbols*. The unparsing rules take one of two forms:

```
phylum : operator [left-side : right-side];  
phylum : operator [left-side ::= right-side];
```

The unparsing scheme between the square brackets represents a concrete display format for the corresponding abstract syntax production. The symbol ::= indicates the production's text is editable, whereas the symbol : indicates the production is (usually) treated as an indivisible structural unit. Not all operators of a given production have to use the same symbol. The *left-side* is a selection symbol (explained below) representing the left-hand-side of the corresponding abstract syntax production. The *right-side* is a sequence of strings, attribute-names and selection symbols representing the right-hand-side of the corresponding abstract syntax production, and defines the display format for that production. [Ref. 10: p. 59]

Indentation and line breaks are indicated by the following control characters:

```
%t - move the left margin one indentation unit to the right  
%b - move the left margin one indentation unit to the left  
%n - break the line and move to the left margin of the next line
```

These control characters can be included in strings (i.e., inside double quotes) of an unparsing rule. [Ref. 10: pp. 59-60]

Each nonterminal (i.e., phylum name) in an abstract syntax production is replaced by the *selection symbol* @ or ^ in the corresponding unparsing rule. These selection symbols define the *selectability property* for the corresponding node in the tree. The selection symbol @ specifies that the phylum occurrence is a *resting place*; the selection symbol ^ denotes a non-resting place. A resting place is a point where the corresponding phylum occurrence can be expanded through template insertion or text entry. Each node in the syntax-tree is an instance of *two* phylum occurrences in the grammar: as a left-hand-side occurrence in one production, and as part of a right-hand-side occurrence in another. If either of its two corresponding occurrences is specified with an @, that node is a resting place. [Ref. 10: p. 61]

Care must be taken when defining unparsing schemes for list phyla. The two occurrences of the list phylum (i.e., the left-hand-side and the second argument on the right-hand-side) should be defined with @ and the element-phylum (the first argument on the right-hand-side) should be defined with ^. This prevents an extra resting place at the element-phylum position. *Conditional unparsing* of list separators are declared inside square brackets. These list separators are suppressed from the display of production instances occurring at the end of a list. For example, in Figure 9, the ["%n%n"] in the unparsing declaration for ModulePair cause each element of the list to be separated by a two line-feeds. [Ref. 10: p. 61]

The unparsing declarations should initially contain the maximal number of resting places to allow full exploration and debugging of the abstract syntax. Plan on eliminating undesirable resting places later by changing some of the phyla occurrences from @ to ^. [Ref. 10: p. 164]

Primitive (lexical) phyla occurrences are an exception to this rule--they should *not* be resting places. This has the effect of forcing the resting place one node higher in the tree. To see why this is useful, imagine the following situation based on the abstract syntax and unparsing rules for a desk calculator depicted in Figure 10. Suppose the selection is positioned at an occurrence of phylum INT in one of the desk calculator's expressions. The pre-defined primitive phylum INT has a placeholder term of 0, which would replace the selection if the command **delete-selection** <^K> were executed and INT was a resting place. But if INT is not a resting place, the selection is forced one level higher to Const(INT). Now if the command **delete-selection** is executed, the placeholder term for exp, namely Null, replaces the selection. The display representation for Null is the string <exp>. [Ref. 10: pp. 163-165]

Since occurrences of primitive phyla are not resting places, in operators of arity two or more, the primitive phyla occurrences will not be individually selectable. Therefore, an extra syntactic level is necessary to allow the individual selection of components. Failure to add this extra level is a common mistake made when designing an editor specification. [Ref. 10: p. 165]

start	Spec	[@ : @]
spec	ModuleNil	[@ ::=]
	ModulePair	[@ ::= ^ ["%n%n"] @]
module	DefModuleDecl	[@ ::= "DEFINITION " @ @ "%n" "END"]
interface	InterfaceDecl	[@ ::= @ "%t%n" @ "%n" @ "%n" @ "%b%n"]
where	WhereEmpty	[@ ::=]
	WhereExp	[@ ::= "WHERE " @]
concepts	ConceptNil	[@ ::=]
	ConceptPair	[@ ::= ^ ["%n"] @]
concept	ConceptType	[@ ::= "%t%n""CONCEPT " @ " : " @ "%t%n" @ "%b%n"]
	ConceptValue	[@ ::= "%t%n""CONCEPT " @ @ "%t%n" @ "%n" "VALUE " @ "%n" @ "%b%n"]
formal_name	FormName	[@ ::= @ " " @]
expression_list	SingleExp	[@ ::= @]
	MultiExp	[@ ::= @ ", " @]
expression	UndefExp	[@ ::= "?"]
	Equal	[@ ::= @ " = " @]
	Add	[@ ::= @ " + " @]
	Subtract	[@ ::= @ " - " @]
identifier	Identifier	[@ ::= ^]

Figure 9
Partial Listing of Spec Unparsing Declarations

As with other kinds of SSL declarations, the phylum name X_0 can be factored to the left when there are multiple transformations on the same phylum [Ref. 11: p. 86]:

```
transform  $X_0$ 
  on transformation-name1 <X0> : operator1 (<X1>, ..., <Xn>),
  on transformation-name2 <X0> : operator2 (<Y1>, ..., <Yn>),
  ...
  on transformation-namek <X0> : operatork (<Z1>, ..., <Zn>)
;
```

The *transformation-name* is enclosed in double quotes and constitutes command that can be invoked to replace the placeholder term with the invoked template (see Appendix C for information on invoking template transformations). Figure 11 is a partial listing of the template transformations for Spec.

List phyla and optional phyla do not normally require template transformations. Commands that move the selection (e.g., **forward-with-optionals** [^M or RETURN] and **forward-sibling-with-optionals** [ESC-^M]) provide adequate transformations for these phyla. [Ref. 10: p. 166]

```
transform concept
  on "type"      <concept>:  ConceptType(<formal_name>,<type_spec>,<where>),
  on "value"     <concept>:  ConceptValue(<formal_name>,<formal_arguments>,<where>,<formal_arguments>,<where>)
;

transform expression_list
  on "single"    <expression_list>:  SingleExp(<expression>),
  on "multiple"  <expression_list>:  MultiExp(<expression_list>,<expression>)
;

transform expression
  on "="         <expression>:  Equal(<expression>,<expression>),
  on "+"         <expression>:  Add(<expression>,<expression>),
  on "-"         <expression>:  Subtract(<expression>,<expression>)
;
```

Figure 11
Partial Listing of Spec Template Transformations

4. Generate and Debug Editor

At this stage the partial editor specification defines enough of the editor's structure-editing facilities to allow the Synthesizer Generator to create an editor and for you to test the editor's characteristics. Invoke `sgen` on the specification files. By default, the created editor will be placed in the file `syn.out`. This can be overridden with the `-o` option which allows the user to declare what filename the editor will be called. [Ref. 10: p. 167]

Recommended options to include at this point are the `-w window-system-name` window option, which specifies that the editor will be created for the specific *window-system-name*, and the `-v` option, which invokes Yacc [Ref. 18] with the `-v` flag so the diagnostic file `y.output` will be produced. This file is useful for debugging the specification file. [Ref. 10: p. 288]

The `y.output` file produced by Yacc is a complete listing of the state machine model that represents the created editor. If there are any parsing conflicts in the editor specification, these can be located in the `y.output` file by loading it into the `vi` editor and doing a search for the word `conflict`. Specifically, the types of conflicts that can occur are *shift/reduce* and *reduce/reduce* conflicts, both of which are described in detail below.

5. Define Concrete Input Syntax

Next, define the concrete syntax for textual input. The concrete syntax consists of lexical declarations, parsing declarations, and correspondence rules for connecting the concrete syntax and the abstract syntax. Initially, provide only those rules needed to

permit entering lexemes and simple expressions. The rules can be elaborated later to permit text entry for additional language constructs. [Ref. 10: p. 167]

The term *text editing* refers to the process of modifying the textual representation of an object through operations on the character sequence [Ref. 10: p. 108]. *Structure editing* is the process of modifying the underlying abstract syntax tree structure of an object [Ref. 10: p. 95].

Text editing is more appropriate than structure editing for entering and modifying expressions. It is also needed for entering identifiers. While a subterm is being edited, the usual rigors of the syntax directed discipline are temporarily suspended, since the subterm being edited can be any string whatsoever. When the editing is complete, the string is parsed and translated to the corresponding term of the abstract syntax. [Ref. 10: p. 62]

a. Parsing Declarations

The *parsing declarations* that define the concrete input syntax are distinguished from the other phylum and operator declarations of an SSL specification by using the symbol ::= instead of : to separate the left-hand-side phylum name from the right-hand-side symbols. Additionally, parsing declarations may have *tokens* (single characters enclosed in quotes) interspersed among the phylum symbols on the right-hand-side.

Each parsing declaration has the form:

phylum-name ::= *operator-name* (*tokens phylum₁ tokens phylum₂ ... phylum_k tokens*);

Some of the *phylum_i* may be lexical phyla that define keywords or other multi-character tokens. Knowing which string occurred in the input is usually relevant. Therefore, the corresponding string appears as the *i*-th argument of the operator in the parse tree. Examples of parsing declarations are shown in Figure 12. [Ref. 11: pp. 76-77]

Attribute equations for constructing terms from text input are combined with the parsing declarations using the following syntax:

phylum-name ::= operator-name (parsing-scheme) {equations};

Synthesized attributes of the left-hand-side phylum and inherited attributes of the right-hand-side phylums are termed the production's *output attributes*. An attribute equation defines the value of an output attribute in terms of other attributes accessible within the production. The attribute equation(s) are grouped inside curly braces. An output attribute *b* of phylum *X* is denoted by *X.b*. If there is more than one occurrence of phylum name *X* in an attribute equation, the various occurrences of *b* are denoted (from left to right) by *X\$1.b*, *X\$2.b*, etc. An attribute *b* of a left-hand-side phylum of a production can be denoted by *\$\$b*. [Ref. 11: pp. 27-28]

Examples of attribute equations are shown in Figure 13. In the attribute equations for SPECDEF, the attribute *t* is for *text*. As an example of how the attribute equations work, let's examine the equation for phylum FormalName. The text attribute for FormalName is given the value formed by applying the abstract syntax operator FormName to the value of the text attributes of the concrete input phyla Ident and FormalParm.

Module	::= (DEFINITION Interface Concepts END);
Interface	::= (FormalName Inherits Imports Export);
...	
Concepts	::= (Concept)
	(Concept Concepts)
	;
...	
FormalName	::= (Ident FormalParm);
FormalParm	::= ()
	('{' FieldList '}' Where)
	;
...	
Ident	::= (IDENTIFIER);

Figure 12
Concrete Syntax Parsing Declarations

The format of attribute equations for list phyla is different. Examine the attribute equations for phylum Concepts in Figure 13. For the first production, the text attribute of Concepts is given the value formed by concatenating the value of the text attribute of phylum Concept with the nullary term ConceptNil. For the second production, the new value of the text attribute of phylum Concepts, designated on the left-hand-side as Concepts\$1.t, is given the value formed by concatenating the value of the text attribute of the phylum Concept with the previous value of the text attribute of phylum Concepts, designated on the right-hand-side as Concepts\$2.t. Recall that multiple occurrences of a phylum name *X* within an attribute equation are differentiated left to right as *X*\$1, *X*\$2, etc.

Module	::= (DEFINITION Interface Concepts END) (Module.t = DefModuleDecl(Interface.t, Concepts.t);)
Interface	::= (FormalName Inherits Imports Export) (Interface.t = InterfaceDecl(FormalName.t, Inherits.t, Imports.t, Export.t);)
...	
Concepts	::= (Concept) (Concepts.t = (Concept.t :: ConceptNil);) (Concept Concepts) (Concepts.\$1.t = (Concept.t :: Concepts.\$2.t);)
...	
FormalName	::= (Ident FormalParm) (FormalName.t = FormName(Ident.t, FormalParm.t);)
FormalParm	::= () (\$\$.t = FormalParmEmpty();) ('(' FieldList ')' Where) (FormalParm.t = FormalParmList(FieldList.t, Where.t);)
...	
Ident	::= (IDENTIFIER) (Ident.t = Identifier(IDENTIFIER);)

Figure 13
Parsing Declarations with Attribute Equations

The operator names of parse tree terms are optional, and usually are not needed since the parser constructs the parse tree. A unique operator name is automatically generated by the parser. [Ref. 11: p. 78]

b. Correspondence Between Concrete Syntax and Abstract Syntax

To translate from text to term, rules are needed that define the association between the abstract syntax and the concrete input syntax, along with attribute equations that synthesize the term as an attribute of the parse tree. These rules are associated with the productions of the concrete syntax, which is thus extended to become an attributed grammar which generates terms of the abstract syntax. [Ref. 10: p. 62]

Attributes must be defined with an *attribute declaration* which has the form
concrete-syntax-phylum { synthesized abstract-syntax-phylum attribute-name; }

Examples of attribute declarations are shown in Figure 14. Note that the reserved word `synthesized` can be abbreviated as `syn`. [Ref. 11: pp. 24-25]

A convenient convention for naming the concrete syntax phylum is to use the same name as the corresponding abstract syntax phylum and capitalize the first letter of each "word" within the name (e.g., `FormalName`). Occasionally you may have to invent some other name if such a capitalized name has already been used, for example, as an operator in the abstract syntax declarations. Attribute names are often abbreviated words or phrases that are descriptive of what kind of attribute they are naming (e.g., `t` for text, `env` for environment, etc.) [Ref. 11: p. 9; 10: p. 63]

Entry declarations establish the correspondence between selections in the abstract syntax and entry points within the concrete input syntax. Each entry declaration has the form:

abstract-syntax-phylum ~ concrete-syntax-phylum.attribute-name;

This declaration specifies that when the current selection in the abstract-syntax tree is a node of phylum *abstract-syntax-phylum*, the input is parsed according to the parse declarations of *concrete-syntax-phylum*, and the value of attribute *concrete-syntax-phylum.attribute-name* is inserted into the abstract-syntax tree, replacing the currently selected subterm or sublist. Figure 14 is a partial listing of text attribute declarations and entry declarations for Spec. [Ref. 10: pp. 62-65; 11: pp. 74-76]

Module	{ syn module t; };
Interface	{ syn interface t; };
...	
Where	{ syn where t; };
Concepts	{ syn concepts t; };
Concept	{ syn concept t; };
FormalName	{ syn formal_name t; };
...	
ExpList	{ syn expression_list t; };
Exp	{ syn expression t; };
Ident	{ syn identifier t; };
module	~ Module.t;
interface	~ Interface.t;
...	
where	~ Where.t;
concepts	~ Concepts.t;
concept	~ Concept.t;
formal_name	~ FormalName.t;
...	
expression_list	~ ExpList.t;
expression	~ Exp.t;
identifier	~ Ident.t;

Figure 14
Association Between Abstract Syntax and Input Syntax

c. *Concrete Lexical Declarations*

Lexical phyla should be declared for each keyword and other multi-character token of the language. A declaration for the special token **WHITESPACE** should be included at this point if it has not been previously declared. Examples of lexical definitions are shown in Figure 15. [Ref. 10: p. 167]

WHITESPACE:	WhiteSpace < [\ \n]* >;
AND:	AndLex < "&" >;
OR:	OrLex < " " >;
NOT:	NotLex < "~" >;
...	
CONCEPTS:	ConceptLex < "CONCEPT" >;
DEFINITION:	DefnLex < "DEFINITION" >;
END:	EndLex < "END" >;
...	
SEMI:	SemiLex < ";" >;
PLUS:	PlusLex < "+" >;
MINUS:	MinusLex < "-" >;
...	
INTEGER_LIT:	IntegerLex < [0-9]+ >;
IDENTIFIER:	IdentLex < [a-zA-Z][a-zA-Z_0-9]* >;

Figure 15
Lexical Definitions

d. Parsing and Ambiguities

The Synthesizer Generator uses the parser generator Yacc [Ref. 18] to create the editor's parser. As the parser scans the input stream, it pushes and pops tokens onto/from a stack according to the parsing declarations. A *shift* pushes the next token α onto the stack; a *reduce* pops tokens off the stack when all tokens necessary to complete a parsing rule have been seen. The resulting token (an occurrence of the left-hand-side of a parsing declaration) is then pushed onto the stack **before** the next token α is parsed.

Yacc will detect and report any ambiguities in the input grammar. Ambiguities are of two types: a *shift/reduce* conflict or a *reduce/reduce* conflict. A *shift/reduce* conflict arises when the parser cannot determine, based on the stack contents and the next token α , whether to apply a reduction rule or to shift α onto the stack, thereby deferring the reduction. A *reduce/reduce* conflict arises when the parser can apply more than one

reduction rule to the current contents of the stack. By default, Yacc resolves these conflicts as follows:

- For a shift/reduce conflict, perform the shift.
- For a reduce/reduce conflict, reduce by the production declared earliest in the specification.

These resolution rules will often not do what you intended. Therefore, some additional mechanism is required to resolve ambiguities. [Ref. 11: p. 79]

Precedence declarations provide this additional mechanism for resolving ambiguities in the input grammar. They must be listed prior to the parsing declarations. These precedence declarations associate precedence levels with characters and lexemes. There are three forms of precedence declarations:

```
left      token-or-phylum1,...,token-or-phylumk;  
right     token-or-phylum1,...,token-or-phylumk;  
nonassoc token-or-phylum1,...,token-or-phylumk;
```

Each *token-or-phylum* can be a single CHAR constant, such as '+', or the name of a lexical phylum, such as IDENTIFIER. The same precedence level and associativity is assigned to all characters and lexemes denoted by *token-or-phylum₁,...,token-or-phylum_k*. The order in which precedence declarations are listed in the SSL specification is extremely important--each successive declaration receives a higher precedence than the previous declaration(s) (i.e., precedences are declared low-to-high). The keywords *left*, *right*, and *nonassoc* define the associativity of the listed tokens. The effects of precedence and associativity are described below. [Ref. 11: p. 79]

The final token in a concrete syntax production determines the precedence level and associativity of the production. Precedences and associativity can be associated

with complete productions as well as with tokens. To assign an explicit level of precedence to a production, which could be either the same as or different than the precedence level of the last token (if one is present at all), the parsing declaration uses the form:

phylum-name ::= operator-name (list-of-tokens-and-phyla prec token-or-phylum);

Yacc uses these precedence rules to disambiguate the productions of the input grammar as follows:

- When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol (i.e., character or lexeme name) or the grammar rule has no precedence and associativity, the disambiguation rules stated above are applied and the conflicts reported.
- If there is a shift/reduce conflict, and both grammar rule and input character have precedence and associativity, the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If precedences are the same, associativity is used: left associative implies reduce; right associative implies shift; nonassociating implies error.

Using the -v option of `sgen` is extremely helpful in debugging such conflicts. The `y.output` file generated by Yacc will indicate all shift/reduce and reduce/reduce conflicts and which tokens caused them. After running `sgen`, edit this file and perform a search for the word "conflict". [Ref. 11: p. 80]

Figure 16 is a partial listing of the lexical phyla, precedence declarations, and concrete syntax parsing declarations for SPECDEF.

```

/* lexical phyla */
WHITESPACE: WhiteSpace<[\ \n]*>;
CONCEPT:  ConceptLex<"CONCEPT">;
DEFINITION: DefnLex<"DEFINITION">;
END:        EndLex<"END">;
VALUE:      ValueLex<"VALUE">;
WHERE:      WhereLex<"WHERE">;
SEMI:       SemiLex<">">;
COMMA:      CommaLex<">">;
PLUS:       PlusLex<"+">;
MINUS:      MinusLex<"-">;

/* precedence and associativity rules */
left  ':' SEMI IDENTIFIER;
left  ':' COMMA;
left  '=';
left  '+' '-' PLUS MINUS;

/* concrete input syntax */
Module      ::= (DEFINITION Interface Concepts END)
               {Module.t = DefModuleDed(Interface.t, Concepts.t);}

Interface    ::= (FormalName Inherits Imports Export)
               {Interface.t = InterfaceDed(FormalName.t, Inherits.t, Imports.t, Export.t);}

...

Where        ::= (prec SEMI)           {$$t = WhereEmpty();}
               | (WHERE ExpList)      {Where.t = WhereExp(ExpList.t);}

Concepts     ::= (Concept*)           {Concepts.t = (Concept.t :: ConceptNil);}
               | (Concepts Concept)  {Concepts.$1.t = (Concept.t :: Concepts.$2.t);}

Concept      ::= (CONCEPT FormalName ':' TypeSpec Where)
               | (CONCEPT FormalName FormalArg Where VALUE FormalArg Where)
               {Concept.t = ConceptType(FormalName.t, TypeSpec.t, Where.t);}
               {Concept.t = ConceptValue(FormalName.t, FormalArg.$1.t, Where.$1.t,
                                           FormalArg.$2.t, Where.$2.t);}

FormalName   ::= (Ident FormalParm)   {FormalName.t = FormName(Ident.t, FormalParm.t);}

...

ExpList      ::= (Exp prec COMMA)      {ExpList.t = SingleExp(Exp.t);}
               | (ExpList ',' Exp prec COMMA) {ExpList.$1.t = MultiExp(ExpList.$2.t, Exp.t);}

Exp          ::= (?)                   {Exp.t =.UndefExp();}
               | (Exp '=' Exp prec '=') {Exp.$1.t = Equal(Exp.$2.t, Exp.$3.t);}
               | (Exp '+' Exp prec PLUS) {Exp.$1.t = Add(Exp.$2.t, Exp.$3.t);}
               | (Exp '-' Exp prec MINUS) {Exp.$1.t = Subtract(Exp.$2.t, Exp.$3.t);}

Ident        ::= (IDENTIFIER) {Ident.t = Identifier(IDENTIFIER);}

```

Figure 16
Lexical Phyla, Precedence and Associativity Rules, and
Concrete Input Syntax Parsing Declarations for SPECDEF

6. Refine the Display Representation

The final step to complete the initial version of the editor specification is to refine the display representation. In Section IV.C.2 we suggested that the unparsing declarations be written with the maximal number of resting places. Any undesirable resting places should now have their selection symbol changed from @ to ^. It is very important that the selection symbols in lists are defined according to the pattern shown in Figure 17 because selections in lists are handled specially. [Ref. 10: p. 171]

<u>abstract syntax rule</u>		<u>unparsing rule</u>
list listType;		
listType	ListTypeNull()	[@ ::=]
	ListTypePair(listElement listType)	[@ ::= ^ @]
listElement	ListElementNull()	[^ ::= "<listElement>"]

Figure 17
Unparsing pattern for list phyla

For example, the unparsing declarations shown in Figure 9 should be changed to have some of the selection symbols on the left-hand-side of the unparsing rules changed from @ to ^ as shown in Figure 18.

Finally, the editing-mode symbols in the unparsing declarations (i.e., the ::= and : symbols) need to be checked that they specify the desired text-editing properties for the various operators. Specifically, if the operator allows text entry, the ::= symbol should be used, but if text entry or text re-editing is not allowed, the : symbol should be used. Note that the symbol ::= also allows textual *re-editing* of elements. Therefore, if the designer wanted to allow the entry of individual lexemes but not full expressions, the unparsing

rules for the phylum expression should be changed from what was shown in Figure 18 to those shown in Figure 19. [Ref. 10: p. 172]

start	Spec	[@ : @]
spec	ModuleNil	[@ ::=]
	ModulePair	[@ ::= ^ ["%n%n"] @]
module	DefModuleDecl	[@ ::= "DEFINITION " @ @ "%n" "END"]
interface	InterfaceDecl	[@ ::= @ "%t%n" @ "%n" @ "%n" @ "%b%n"]
...		
where	WhereEmpty	[^ ::=]
	WhereExp	[^ ::= "WHERE " @]
concepts	ConceptNil	[@ ::=]
	ConceptPair	[@ ::= ^ ["%n"] @]
concept	ConceptType	[^ ::= "%t%n""CONCEPT " @ " : " @ "%t%n" @ "%b%b%n"]
	ConceptValue	[^ ::= "%t%n""CONCEPT " @ @ "%t%n" @ "%n" "VALUE " @ "%n" @ "%b%b%n"]
formal_name	FormName	[^ ::= @ " " @]
...		
expression_list	SingleExp	[^ ::= @]
	MultiExp	[^ ::= @ ", " @]
expression	UndefExp	[^ ::= "?"]
	Equal	[^ ::= @ " = " @]
	Add	[^ ::= @ " + " @]
	Subtract	[^ ::= @ " - " @]
identifier	Identifier	[^ ::= ^]

Figure 18
Refined Unparsing Declarations for Spec

expression	:	UndefExp	[^ ::= "?"]
		Equal	[^ : @ " = " @]
		Add	[^ : @ " + " @]
		Subtract	[^ : @ " - " @]
	:		

Figure 19
Unparsing rules that forbid re-editing

7. Advanced Unparsing Features

We have already discussed the use of the control strings %t, %b and %n in unparsing declarations to signal tab, backtab, and new line, respectively. It should be noted that changing the left-margin (i.e., with %t or %b) has no immediate visible effect. The change is only evident after the start of a new line. [Ref. 11: p. 64]

To prevent the undesirable effect of having long lines of display text broken in mid-word, optional line breaks, indicated by %O, provide the means to specify places where a term's display string may be divided into separate lines. The designer can impose a hierarchical structure on the display string by the matched grouping symbols %{ and %}. The editor will attempt to split long lines containing optional line breaks in such a way that the minimum number of unparsing groupings will be split and the amount of text on the line is maximized. A grouping only has effect if it contains an optional line break. If indentation is desired only for the text that follows an optional line break that is taken, the formatting command string %t%O%i) may be used. A complete listing of unparsing display commands is shown in Figure 20. [Ref. 11: pp. 63-66]

Formatting Command	Meaning
%t	move the left-margin one indentation unit to the right
%b	move the left-margin one indentation unit to the left
%n	break the line and return to the current left-margin
%l	return to current left-margin of the same line and overprint
%1	move to column one of the same line and overprint
%T	move right to the next tab stop
%M(c)	move right to column c, where c is a positive integer
%o	optionally, break the line and return to the current left-margin
%c	same as %o, but either all or no %c in a group are taken
%{	beginning of an unparsing group
%}	end of an unparsing group
%{	same as %t%{
%}	same as %}%b
%S(style-name:	enter the named style
%S)	revert to the previous style
%%	display a %

Figure 20
Formatting Commands

To see how some of these commands can be used, let's look at an example of an unparsing declaration that includes groupings and optional line breaks. The following declaration uses the formatting commands for beginning and end of an unparsing group, beginning and end of an unparsing group with tabs and backtabs, and optional line breaks:

```
expression : QuantifierExp  [^ ::= "%{" @ "(" @ "%o" @ "%}%[%o :: " @ "%}"]
```

The resulting expression could be displayed in several formats that would be dependent on the width of the current window. Figure 21 shows these different formats with one or more of the optional line breaks taken.

```
ALL(item : type1 SUCH THAT foo(a) < 27 :: item = *item + 1)
```

```
ALL(item : type1  
SUCH THAT foo(a) < 27 :: item = *item + 1)
```

```
ALL(item : type1 SUCH THAT foo(a) < 27  
:: item = *item + 1)
```

```
ALL(item : type1  
SUCH THAT foo(a) < 27  
:: item = *item + 1)
```

Figure 21
Effect of optional line breaks on display

D. DESIGN DECISIONS

1. Spec Subset

The subset of Spec I chose to implement is the Definition Module. Definition modules are used in the requirements analysis phase of the software development cycle [Ref. 1]. Implementing an editor for the definition module provides the initial support for utilizing Spec in this phase of software engineering.

2. Type Checking

Editors created by the Synthesizer Generator can have the capability to perform type checking provided the proper attribute equations are written to support this checking. However, since other research is currently being done to implement a separate type checker for Spec [Ref. 15], the decision was made to not implement this capability in the syntax directed editor. Eventually, when the editor and the type checker are fully implemented, both tools will be integrated into one larger tool that will perform both functions.

3. Display Representation

In order to make the visual display "user friendly", many of the constructs in Spec had productions added whose purpose is to display a placeholder string. These productions appear in both the abstract syntax and the unparsing grammar. The format of these placeholder strings is the phylum name enclosed in angle brackets, e.g., <concept> or <actual_parm>.

Although not part of the original Spec grammar, these productions are declared as the completing term for the respective phylum. The phyla that have this additional production are all non-list phyla, therefore the completing term is also the placeholder term for the given phylum. This causes the visual display to show where a selection should be made at a later time if a template transformation is not selected.

Some constructs in Spec already have a usable completing term, specifically field, type_spec, and expression, which all use the symbol ? to represent undefined occurrences of the respective phylum.

4. Naming Conventions

The following conventions were selected for naming of phyla, operators, and attributes:

- Abstract syntax phyla are all lower case (e.g., concept), and are exactly the same as the original Spec grammar.
- Operators on abstract syntax phyla have the first letter of each "word" within the operator name capitalized (e.g., NotGreaterEqual).
- Lexemes are fully capitalized (e.g., IDENTIFIER).

- Lexical phyla operators are capitalized with the same conventions as abstract syntax phyla operators. Additionally, with the exception of the special lexical phylum `WHITESPACE`, each of these operators ends in the word `Lex` (e.g., `NgeLex`).
- Template transformation names are all lower case (e.g., `"single"`), or are a symbol made of one or more non-alphanumeric characters (e.g., `"+"` or `"~>="`).
- Concrete input syntax phyla have the same name as the corresponding abstract syntax phyla (with only a few exceptions) with the first letter of each "word" within the name capitalized (e.g., `FormalName` or `IntegerLit`, corresponding to abstract syntax phyla `formal_name` and `integer_lit`, respectively).
- Attributes are all labelled as a lower case `t`, since the only attributes used in this editor are synthesized text attributes.

5. Implementation

A listing of the current implementation of SPECDEF is given in Appendix B. A user's manual for the current version of SPECDEF appears as Appendix C.

V. CONCLUSIONS

A. APPLICABILITY OF THE SYNTAX DIRECTED EDITOR FOR SPEC

The SPECDEF editor successfully demonstrates the feasibility of designing a CASE tool for the specification language Spec. The current version of the editor supports a subset of Spec consisting solely of definition modules and prevents the user from writing syntactically incorrect specifications. All Spec constructs that are applicable to a definition module are implemented in the editor. The SPECDEF editor provides the means to quickly and correctly write Spec definition modules in support of the requirements analysis phase of the software development cycle.

The display representation is designed in such a way that the editor can assist the novice user in learning the syntax of Spec. As with other syntax directed editors for other languages, this could be seen as a hindrance or an annoyance to the more experienced Spec specification writer, since the method of building a parse tree of the specification being edited requires the user to traverse the entire parse tree of the Spec grammar itself.

A significant foundation has been laid for designing an editor that will implement the entire Spec language. Approximately half of the Spec grammar rules have been implemented.

B. USEFULNESS OF THE SYNTHESIZER GENERATOR

The Synthesizer Generator proved to be an extremely useful, albeit sometimes difficult, CASE tool to assist in the design and implementation of a syntax directed editor. The book and reference manual for the Synthesizer Generator [Ref. 10 and 11] are not always clear in their explanations of the various concepts that must be mastered to write a successful editor specification.

I found it very advantageous to refer to the specification of the `toy.syn` editor, one of the sample editors that comes with the `sgen` system. The specification file `toy.ssi` can be found in the `/usr/suns2/local/syn/editors/toy` directory on the `suns2` system. There are several other sample editors in the `/editors` subdirectory of `/syn` whose specification files may provide insight to some readers.

The various options available when invoking `sgen` make the Synthesizer Generator system very versatile. Specifically, the `-v` option to flag Yacc to produce the diagnostic file `y.output` is extremely useful in debugging ambiguities in the input grammar. The ability of `sgen` to create an editor for different windowing systems using the `-w` option makes this system very powerful, since the same input file(s) can be used to create many different editors that target specific systems.

The Synthesizer Generator creates many intermediate files during the process of generating an editor. Unless you have prodigious amounts of free memory available, I highly recommend that the `-l` option (lower case L) *not* be invoked. This option saves all intermediate files, which can easily use up several megabytes of memory. Under normal operation these files are deleted once the executable editor file is generated.

Learning the editor system commands to navigate around the parse tree of a specification being edited is rather difficult. There is quite a list of commands available; a complete list appears as Appendix D. However, only a portion of the commands are routinely used to edit a specification, as listed in Table 2. The use of these commands is discussed in the user's manual, which appears as Appendix C. Once these commands are learned, writing a Spec definition module is quick and easy.

Table 2: COMMONLY USED EDITOR COMMANDS

<u>Command</u>	<u>Key-binding</u>
backward-preorder	< ^P >
beginning-of-line	< ^A >
delete-next-character	< ^D >
delete-other-windows	< ^X1 >
delete-previous-character	< DEL >
delete-selection	< ^K >
end-of-file	< ESC-> >
end-of-line	< ^E >
enlarge-help	< ESC-^Xz >
erase-to-beginning-of-line	< ESC-DEL >
erase-to-end-of-line	< ESC-d >
execute-command	< ^I or TAB >
exit	< ^C >
forward-preorder	< ^N >
forward-with-optionals	< ^M or RETURN >
left	< ^B >
next-window	< ^Xn >
previous-window	< ^Xp >
read-file	< ^X^R >
redraw-display	< ^L >
right	< ^F >
select	< ESC-@ >
start-command	< ESC-s >
write-named-file	< ^X^W >

C. DEFICIENCIES AND BUGS ENCOUNTERED

The SPECDEF editor is strictly a syntax directed editor. There is no capability provided for semantic checking. As stated previously, the type checking features that **sgen** is capable of producing are not included in this editor since independent research in this area is currently in progress. This means that, although a specification written using the editor will be syntactically correct, it is possible for it to be incorrect semantically.

One deficiency with the SPECDEF editor is the lack of support to insert comments into the specification being edited. Although **sgen** is capable of supporting this feature when properly specified in the editor specification, time constraints prevented the inclusion of this capability, which requires extensive coding to accomplish.

D. RECOMMENDATIONS FOR FUTURE RESEARCH

1. Implement the Complete Spec Language

The remainder of the Spec language needs to be implemented in the editor. When this is accomplished, the editor will support not only the requirements analysis phase of the software development cycle, but also the functional specification and architectural design phases.

2. Integrate the Type Checker and the Editor

The type checker currently being implemented independently should be integrated with the editor once both tools are completed. This will make the editor a very powerful tool for software engineering with Spec. The combined tool will form the foundation of an integrated Spec programming environment.

3. Integrated Spec Programming Environment

Many different tools for Spec are under development or planned for future research. A pretty printer has already been designed. The type checker is currently under development. Future research projects include a consistency checker, an inheritance expander, a test oracle, and a diagram generator. There are also plans to design a translator from Spec specifications to compilable Ada code.

Each of the above mentioned tools, along with the editor, are planned for integration into a programming environment for Spec. This environment will be capable of supporting the entire software development cycle.

4. Investigate Applicability of Utilizing the GANDALF System

The GANDALF System [Ref. 7] might be more useful than the Synthesizer Generator for creating a syntax directed editor. Some of the additional capabilities of GANDALF are the ability to generate system environments that could support multiple programmers on large projects, and a limited project management system. The need for these capabilities should be investigated, and if it is determined that such a need exists, future versions of the editor should be built using GANDALF instead of the Synthesizer Generator.

5. Automating the Design of an SSL Specification

After the conceptual breakthrough occurs on how to design an editor specification using SSL, adding more constructs to the grammar becomes a relatively mechanical process. The different sections of the SSL specification are closely related in terms of names of phyla and operators. This indicates a possibility that some aspects of designing

an SSL editor specification could be automated. For example, when an abstract syntax rule is entered, the automation process could generate the display representation rules, template transformation rules, etc., possibly through the use of pop-up windows that request specific information for naming or formatting the additional rules, or asking for information about attributes.

Automating some portions of this process would enhance the usability of a tool such as the Synthesizer Generator. Additionally, the amount of time to produce an editor specification would be reduced dramatically.

APPENDIX A

COMPLETE SPEC GRAMMAR

The following is a listing of the complete grammar for the Spec specification language. The current version of Spec is 1.11. Optimization of the language is still under research, so the reader can expect further revisions to Spec in the future.

```
% version stamp $Header: spec.k,v 1.11 89/04/05 14:02:21 berzins Locked $

! In the grammar, comments go from a "!" to the end of the line.
! Terminal symbols are entirely upper case or enclosed in single quotes ('').
! Nonterminal symbols are entirely lower case.
! Lexical character classes start with a captial letter and are enclosed in {}.
! In a regular expression, x+ means one or more x's.
! In a regular expression, x* means zero or more x's.
! In a regular expression, [xyz] means x or y or z.
! In a regular expression, [^xyz] means any character except x or y or z.
! In a regular expression, [a-z] means any character between a and z.
! In a regular expression, . means any character except newline.

! definitions of lexical classes

%define      Digit      :{0-9}
%define      Int        :{Digit}+
%define      Letter     :{a-zA-Z}
%define      Alpha      :({Letter}|{Digit}){"_"}
%define      Blank      :{ \t\n}
%define      Quote      :{"}
%define      Backslash   :{"\\"}
%define      Char       :({^"\\"}|{Backslash}{Quote}|{Backslash}{Backslash})

! definitions of white space and comments

      :{Blank}+
      :"--".*"\\n"

! definitions of compound symbols and keywords

AND      : "&"
OR       : "|"
NOT      : "~"
IMPLIES  : "=>"
IFF      : "<=>"
```

LE	: "<="
GE	: ">="
NE	: "~="
NLT	: "~<"
NGT	: "~>"
NLE	: "~<="
NGE	: "~>="
EQV	: "=="
NEQV	: "~=="
RANGE	: "..."
APPEND	: " "
MOD	: {Backslash} MOD
EXP	: "****"
BIND	: "::"
ARROW	: "->"
IF	: IF
THEN	: THEN
ELSE	: ELSE
IN	: IN
U	: U
ALL	: ALL
SOME	: SOME
NUMBER	: NUMBER
SUM	: SUM
PRODUCT	: PRODUCT
SET	: SET
MAXIMUM	: MAXIMUM
MINIMUM	: MINIMUM
UNION	: UNION
INTERSECTION	: INTERSECTION
SUCH	: SUCH(Blank)*THAT
ELSE_IF	: ELSE(Blank)*IF
AS	: AS
CHOOSE	: CHOOSE
CONCEPT	: CONCEPT
DEFINITION	: DEFINITION
DELAY	: DELAY
DO	: DO
END	: END
EXCEPTION	: EXCEPTION
EXPORT	: EXPORT
FI	: FI
FOREACH	: FOREACH
FROM	: FROM
FUNCTION	: FUNCTION
GENERATE	: GENERATE
HIDE	: HIDE
IMPORT	: IMPORT
INHERIT	: INHERIT
INITIALLY	: INITIALLY
INSTANCE	: INSTANCE
INVARIANT	: INVARIANT
MACHINE	: MACHINE
MESSAGE	: MESSAGE
MODEL	: MODEL
OD	: OD

```

OF                :OF
OPERATOR          :OPERATOR
OTHERWISE        :OTHERWISE
PERIOD           :PERIOD
RENAME           :RENAME
REPLY            :REPLY
SEND             :SEND
STATE            :STATE
TEMPORAL         :TEMPORAL
TIME             :TIME
TO               :TO
TRANSACTION      :TRANSACTION
TRANSITION       :TRANSITION
TYPE             :TYPE
VALUE            :VALUE
VIRTUAL          :VIRTUAL
WHEN             :WHEN
WHERE            :WHERE

INTEGER_LITERAL   :{Int}
REAL_LITERAL      :{Int}" Cant {Int}
CHAR_LITERAL      :{" Cant "
STRING_LITERAL    :{Quote}{Char}*{Quote}

NAME              :{Letter}{Alpha}*

! operator precedences
! %left means 2+3+4 is (2+3)+4.

%left            '<', '>', '=', LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV;
%left            ',', COMMA;
%left            SUCH;
%left            IFF;
%left            IMPLIES;
%left            OR;
%left            AND;
%left            NOT;
%left            '<', '>', '=', LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV;
%nonassoc        IN, RANGE;
%left            U, APPEND;
%left            '+', '-', PLUS, MINUS;
%left            '*', '/', MUL, DIV, MOD;
%left            UMINUS;
%left            EXP;
%left            '$', '[', '(', '{', '.', DOT, WHERE;
%left            STAR;

%%
!attribute declarations

%%
! productions of the grammar

start
    : spec
      { }
    ;

```

```

spec
: spec module
{ }
|
{ }
;
! A production with nothing after the "|" means the empty string
! is a legal replacement for the left hand side.

module
: function
{ }
| machine
{ }
| type
{ }
| definition
{ }
| instance ! of a generic module
{ }
;

function
: optionally_virtual FUNCTION interface messages concepts END
{ }
;
! Virtual modules are for inheritance only, never used directly.

machine
: optionally_virtual MACHINE interface state messages transactions
temporals concepts END
{ }
;

type
: optionally_virtual TYPE interface model messages transactions
temporals concepts END
{ }
;

definition
: DEFINITION interface concepts END
{ }
;

instance
: INSTANCE formal_name '=' actual_name END
{ }
| INSTANCE foreach actual_name END
{ }
;
! For making instances or partial instantiations of generic modules.
! The foreach clause allows defining sets of instances.

```

```

interface
: formal_name inherits imports export
{ }
/
! This part describes the static aspects of a module's interface.
! The dynamic aspects of the interface are described in the messages.
! A module is generic iff it has parameters.
! The parameters can be constrained by a WHERE clause.
! A module can inherit the behavior of other modules.
! A module can import concepts from other modules.
! A module can export concepts for use by other modules.

inherits
: inherits INHERIT actual_name hide renames
{ }
/
! Ancestors are generalizations or simplified views of a module.
! A module inherits all of the behavior of its ancestors.
! Hiding a message or concept means it will not be inherited.
! Inherited components can be renamed to avoid naming conflicts.

hide
: HIDE name_list
{ }
/
! Useful for providing limited views of an actor.
! Different user classes may see different views of a system.
! Messages and concepts can be hidden.

renames
: renames RENAME NAME AS NAME
{ }
/
! Renaming is useful for preventing name conflicts when inheriting
! from multiple sources, and for adapting modules for new uses.
! The parameters, model and state components, messages, exports,
! and concepts of an actor can be renamed.

imports
: imports IMPORT name_list FROM actual_name
{ }
/

export
: EXPORT name_list
{ }
/

```

```

messages
    : messages message
      { }
    ;

message
    : MESSAGE formal_message operator response
      { }
    ;

response
    : response_body
      { }
    | response_cases
      { }
    ;

response_cases
    : WHEN expression_list response_body response_cases
      { }
    | OTHERWISE response_body
      { }
    ;

response_body
    : choose reply sends transition
      { }
    ;

choose
    : CHOOSE '(' then_list restriction ')'
      { }
    ;

reply
    : REPLY actual_message where
      { }
    | GENERATE actual_message where      ! used in generators
      { }
    ;

sends
    : sends send
      { }
    ;

send
    : optional_foreach SEND actual_message TO actual_name where
      { }
    ;

```

```

transition
    : TRANSITION expression_list      ! for describing state changes
    { }
    ;

formal_message
    : optional_exception optional_formal_name formal_arguments
    { }
    ;

actual_message
    : optional_exception optional_actual_name formal_arguments
    { }
    ;

where
    : WHERE expression_list
    { }
    %prec SEMI ! must have a lower precedence than WHERE
    ;

optionally_virtual
    : VIRTUAL
    { }
    ;

optional_exception
    : EXCEPTION
    { }
    %prec SEMI
    ;

operator
    : OPERATOR operator_list
    { }
    ;

optional_foreach
    : foreach
    { }
    ;

foreach
    : FOREACH '(' field_list restriction ')'
    { }
    ;
    ! foreach is used to describe a set of messages or instances

```

```

concepts
: concepts concept
  { }
;

concept
: CONCEPT formal_name ':' type_spec where
  ! constants
  { }
: CONCEPT formal_name formal_arguments where VALUE formal_arguments where
  ! functions, defined with preconditions and postconditions
  { }
;

model
      ! data types have conceptual models for values
: MODEL formal_arguments invariant
  { }
;

state
      ! machines have conceptual models for states
: STATE formal_arguments invariant initially
  { }
;

invariant
! invariants are true for all states or instances
: INVARIANT expression_list
  { }
;

initially
! initial conditions are true only at the beginning
: INITIALLY expression_list
  { }
;

transactions
: transactions transaction
  { }
;

transaction
: TRANSACTION formal_name '=' action_list where
  { }
;
! Transactions are atomic.
! The where clause can specify timing constraints.

action_list
: action_list ';' action %prec SEMI ! sequence
  { }
: action
  { }
;

```

```

action
: action action    %prec STAR    ! unordered set of actions
  { }
! IF alternatives FI          ! choice
{ }
! DO alternatives OD          ! repeated choice
{ }
! actual_name                ! a normal message or subtransaction
{ }
! EXCEPTION actual_name      ! an exception message
{ }
;

alternatives
: alternatives OR guard action_list
  { }
! guard action_list
  { }
;

guard
: WHEN expression ARROW
  { }
{ }
;

temporals
: temporals temporal
  { }
{ }
;

temporal
: TEMPORAL NAME where response
  { }
;
! Temporal events are trigged at absolute times,
! in terms of the local clock of the actor.
! The "where" describes the triggering conditions
! in terms of TIME, PERIOD, and DELAY.

optional_formal_name
: formal_name
  { }
{ }
;

formal_name
: NAME formal_parameters
  { }
;

formal_parameters
! parameter values are determined at specification time
: '[' field_list ']' where
  { }
{ }
;

```

```

formal_arguments      ! arguments are evaluated at run-time
    : '(' field_list ')'
    { }
    ;

field_list
    : field_list ',' field
    { }
    | field
    { }
    ;

field
    : name_list ':' type_spec
    { }
    | '$' NAME ':' type_spec
    { }
    | '?'
    { }
    ;

type_spec
    : actual_name      ! name of a data type
    { }
    | '?'
    { }
    ;

name_list
    : name_list NAME
    { }
    | NAME
    { }
    ;

optional_actual_name
    : actual_name
    { }
    ;

actual_name
    : NAME actual_parameters
    { }
    ;

actual_parameters      ! parameter values are determined at specification time
    : '(' arg_list ')'
    { }
    | '%' SEMI ! must have a lower precedence than '('
    { }
    ;

```

```

actual_arguments      ! arguments are evaluated at run-time
: '(' arg_list ')'
{ }
|          %prec SEMI ! must have a lower precedence than '('
{ }
;

arg_list
: arg_list ',' arg      %prec COMMA
{ }
| arg
{ }
;

arg
: expression
{ }
| pair
{ }
;

expression_list
: expression_list ',' expression      %prec COMMA
{ }
: expression                      %prec COMMA
{ }
;

expression
: quantifier '(' field_list restriction BIND expression ')'
{ }
| actual_name actual_arguments
{ }
| actual_name '@' actual_name actual_arguments
{ }
: NOT expression                      %prec NOT
{ }
: expression AND expression           %prec AND
{ }
: expression OR expression           %prec OR
{ }
: expression IMPLIES expression      %prec IMPLIES
{ }
: expression IFF expression          %prec IFF
{ }
: expression '<' expression           %prec LE
{ }
: expression '>' expression           %prec LE
{ }
: expression '=' expression          %prec LE
{ }
: expression LE expression           %prec LE
{ }
: expression GE expression           %prec LE
{ }
: expression NE expression           %prec LE
{ }
: expression NLT expression          %prec LE
{ }
: expression NGT expression          %prec LE
{ }

```

```

| expression NLE expression      %prec LE
| { }
| expression NGE expression      %prec LE
| { }
| expression EQV expression      %prec LE
| { }
| expression NEQV expression     %prec LE
| { }
| '-' expression                 %prec UMINUS
| { }
| expression '+' expression      %prec PLUS
| { }
| expression '-' expression      %prec MINUS
| { }
| expression '*' expression      %prec MUL
| { }
| expression '/' expression      %prec DIV
| { }
| expression MOD expression      %prec MOD
| { }
| expression EXP expression      %prec EXP
| { }
| expression U expression        %prec U
| { }
| expression APPEND expression   %prec APPEND
| { }
| expression IN expression       %prec IN
| { }
| '*' expression                 %prec STAR
| ! *x is the value of x in the previous state
| { }
| '$' expression                 %prec DOT
| ! $x represents a collection of items rather than just one
| ! s1 = {x, $s2} means s1 = union({x}, s2)
| ! s1 = {x, $s2} means s1 = append({x}, s2)
| { }
| expression RANGE expression    %prec RANGE
| ! x in [a .. b] iff x in {a .. b} iff a <= x <= b
| ! [a .. b] is sorted in increasing order
| { }
| expression '.' NAME            %prec DOT
| { }
| expression '[' expression ']'  %prec DOT
| { }
| '(' expression ')'
| { }
| '(' expression NAME ')'        ! expression with units of measurement
| ! standard time units: NANOSEC MICROSEC MILLISEC SECONDS MINUTES HOURS
| ! DAYS WEEKS
| { }
| TIME      ! The current local time, used in temporal events
| { }
| DELAY      ! The time between the triggering event and the response
| { }
| PERIOD      ! The time between successive events of this type
| { }
| literal
| { }
| literal '@' actual_name        ! literal with explicit type
| { }

```

```

| '?'      ! An undefined value to be specified later
| {}
| '!'      ! An undefined and illegal value
| {}
| IF expression THEN expression middle_cases ELSE expression FI
| {}
;

middle_cases
: middle_cases ELSE_IF expression THEN expression
| {}
| {}
;

quantifier
: ALL
| {}
| SOME
| {}
| NUMBER
| {}
| SUM
| {}
| PRODUCT
| {}
| SET
| {}
| MAXIMUM
| {}
| MINIMUM
| {}
| UNION
| {}
| INTERSECTION
| {}
;

restriction
: SUCH expression
| {}
| {}
;

literal
: INTEGER_LITERAL
| {}
| REAL_LITERAL
| {}
| CHAR_LITERAL
| {}
| STRING_LITERAL
| {}
| '#' NAME ! enumeration type literal
| {}
| '[' expressions ']' ! sequence literal
| {}
| '{' expressions '}' ! set literal
| {}

```

```

    | '{' expressions ';' expression '}'          ! map literal
    | '{' }
    | '[' pair_list ']'                          ! tuple literal
    | '{' }
    | '{' NAME BIND expression '}'              ! union literal
    | '{' }
;
! relation literals are sets of tuples

expressions
: expression_list
{ }
|
{ }
;

pair_list
: pair_list ',' pair
{ }
| pair
{ }
;

pair
: name_list BIND expression
{ }
;

operator_list
: operator_list operator_symbol
{ }
operator_symbol
{ }
;

operator_symbol
: NOT
{ }
AND
{ }
OR
{ }
IMPLIES
{ }
IFF
{ }
'<'
{ }
'>'
{ }
'='
{ }
LE
{ }
GE
{ }
NE
{ }
NLT
{ }

```

```

/ NGT
{ }
| NLE
{ }
| NGE
{ }
| EQV
{ }
| NEQV
{ }
| '+'
{ }
| '-'
{ }
| '*'
{ }
| '/'
{ }
| MOD
{ }
| EXP
{ }
| U
{ }
| APPEND
{ }
| IN
{ }
| RANGE
{ }
| ''
{ }
| ''
{ }
/

```

APPENDIX B

I. SPECDEF SSL SPECIFICATION FILES

The following are the SSL specification files for the current version of the SPECDEF editor. There are five modules broken down as follows: abstract syntax and template transformations, lexemes, attribute declarations, concrete input syntax, and unparsing schemes.

A. ABSTRACT SYNTAX AND TEMPLATE TRANSFORMATION RULES

The file *abstract.ssl* contains the abstract syntax grammar and the template transformation rules on that grammar for the SPECDEF editor.

```
/* File: ABSTRACT.SSL */
/*
/*      This file contains the abstract syntax for the Definition Module
/*      subset of the Spec specification language for the SPECDEF editor.
/*      It also contains the template transformations on the abstract
/*      syntax productions.
/*
/* abstract syntax */
root start;
start      :      Spec(spec)
            ;
opt: al list spec;
spec       :      ModuleNil()          /* completing term */
              ModulePair(module spec) /* allows addition of
                                      other modules */
            ;
module     :      DefModuleDecl(interface concepts)
              /* Note: Spec grammar has been modified for this
                  editor to go directly to the Definition
                  Module declaration from Module */
            ;
interface  :      InterfaceDecl(formal_name inherits imports export)
            ;
optional inherits;
inherits   :      InheritNil()
              InheritPrompt()          /* placeholder term */
              InheritStmt(inherits actual name hide renames)
            ;
```

```

optional hide;
hide      :      HideNil()
           |      HidePrompt()          /* placeholder term */
           |      HideList(name_list)
           ;

optional renames;
renames   :      RenameNil()
           |      RenamePrompt()        /* placeholder term */
           |      RenameStmt(renames identifier identifier)
           ;

optional imports;
imports   :      ImportNil()
           |      ImportPrompt()        /* placeholder term */
           |      ImportStmt(imports name_list actual_name)
           ;

optional export;
export    :      ExportNil()
           |      ExportPrompt()        /* placeholder term */
           |      ExportList(name_list)
           ;

optional where;
where     :      WhereEmpty()
           |      WherePrompt()        /* placeholder term */
           |      WhereExp(expression_list)
           ;

optional list concepts;
concepts  :      ConceptNil()
           |      ConceptPair(concept concepts)
           ;

concept   :      ConceptComp()          /* completing term */
           |      ConceptType(formal_name type_spec where)
           |      ConceptValue(formal_name formal_arguments where formal_arguments
where)
           ;

formal_name :      FormName(identifier formal_parameters)
           ;

optional formal_parameters;
formal_parameters :      FormalParmEmpty()
           |      FormalParmPrompt()    /* placeholder term */
           |      FormalParmList(field_list where)
           ;

optional formal_arguments;
formal_arguments :      FormalArgEmpty()
           |      FormalArgPrompt()    /* placeholder term */
           |      FormalArgList(field_list)
           ;

field_list :      FieldListComp()      /* completing term */
           |      SingleField(field)
           |      MultipleField(field list field)
           ;

field      :     .UndefField()          /* completing term */
           |      FieldNameList(name_list type_spec)
           |      FieldCollection(identifier type_spec)
           ;

type_spec  :     .UndefType()          /* completing term */
           |      NamedType(actual_name)
           ;

name_list  :      NameIdent(identifier)
           |      NamePair(name_list identifier)
           ;

```

```

actual_name      :      ActualParmList(identifier actual_parameters)
;
optional actual_parameters;
actual_parameters :      ActParmEmpty()
;
;      ActParmPrompt()      /* placeholder term */
;      ListOfArgs(arg_list)
;
optional actual_arguments;
actual_arguments :      ActualArgNil()
;
;      ActualArgPrompt()      /* placeholder term */
;      ActualArgList(arg_list)
;
arg_list         :      ArgListEmpty()      /* completing term */
;
;      SingleArg(arg)
;      MultiArg(arg_list arg)
;
arg              :      ArgComp()      /* completing term */
;
;      ArgExp(expression)
;      ArgPair(pair)
;
expression_list  :      EmptyExpList()      /* completing term */
;
;      SingleExp(expression)
;      MultiExp(expression_list expression)
;
expression       :      UndefExp()      /* completing term */
;
;      QuantifierExp(quantifier field_list restriction expression)
;      IdentExp(actual_name actual_arguments)
;      AtExp(actual_name actual_name actual_arguments)
;      NotExp(expression)
;      AndExp(expression expression)
;      OrExp(expression expression)
;      ImpliesExp(expression expression)
;      IffExp(expression expression)
;      LessThan(expression expression)
;      GreaterThan(expression expression)
;      Equal(expression expression)
;      LessEqual(expression expression)
;      GreaterEqual(expression expression)
;      NotEqual(expression expression)
;      NotLessThan(expression expression)
;      NotGreaterThan(expression expression)
;      NotLessEqual(expression expression)
;      NotGreaterEqual(expression expression)
;      Equivalent(expression expression)
;      NotEquivalent(expression expression)
;      UnaryMinus(expression)
;      Add(expression expression)
;      Subtract(expression expression)
;      Multiply(expression expression)
;      Divide(expression expression)
;      ModExp(expression expression)
;      Exponent(expression expression)
;      UExp(expression expression)
;      AppendExp(expression expression)
;      InExp(expression expression)
;      Star(expression)
;      CollectExp(expression)
;      RangeExp(expression expression)
;      DotExp(expression identifier)
;      SqBracketExp(expression expression)
;      ParenExp(expression)

```

```

| MeasureExp(expression identifier)
| TimeExp()
| DelayExp()
| PeriodExp()
| Constant(literal)
| LiteralType(literal actual_name)
| IllegalExp()
| IfThen(expression expression middle_cases expression)
;

optional middle_cases;
middle_cases : MidCaseNil()
| MiddlePrompt() /* placeholder term */
| MidCase(middle_cases expression expression)
;

quantifier : QuantifierComp() /* placeholder term */
| All()
| Some()
| Number()
| Sum()
| Product()
| Set()
| Maximum()
| Minimum()
| Union()
| Intersection()
;

restriction : RestrictComp() /* placeholder term */
| RestrictNil()
| SuchThat(expression)
;

literal : EmptyLiteral() /* placeholder term */
| IntLiteral(integer_lit)
| RealLiteral(real_lit)
| CharLiteral(char_lit)
| StringLiteral(string_lit)
| Enumeration(identifier)
| Sequence(expressions)
| SetLiteral(expressions)
| MapLiteral(expressions expression)
| TupleLiteral(pair_list)
| OneOfLiteral(pair)
;

expressions : ExpressionsComp() /* placeholder term */
| ExpressionsNil()
| ExpressionsList(expression_list)
;

pair_list : PairListComp() /* placeholder term */
| SinglePair(pair)
| MultiPair(pair_list pair)
;

pair : PairBind(name_list expression)
;

integer_lit : IntConstant(INTEGER_LIT)
;

real_lit : RealConstant(REAL_LIT)
;

char_lit : CharConstant(CHAR_LIT)
;

```

```

string_lit      :      StringConstant (STRING_LIT)
;
identifier      :      Identifier (IDENTIFIER)
;

/* template commands */
transform inherits
    on "empty"      <inherits>:      InheritNil(),
    on "addinherit" <inherits>:      InheritStmt (<inherits>, <actual_name>,
                                                <hide>, <renames>)
;
transform hide
    on "empty"      <hide>:          HideNil(),
    on "addhide"    <hide>:          HideList (<name_list>)
;
transform renames
    on "empty"      <renames>:      RenameNil(),
    on "addrename"  <renames>:      RenameStmt (<renames>, <identifier>, <identifier>)
;
transform imports
    on "empty"      <imports>:      ImportNil(),
    on "addimport"  <imports>:      ImportStmt (<imports>, <name_list>, <actual_name>)
;
transform export
    on "empty"      <export>:      ExportNil(),
    on "addexport"  <export>:      ExportList (<name_list>)
;
transform where
    on "empty"      <where>:        WhereEmpty(),
    on "addwhere"   <where>:        WhereExp (<expression_list>)
;
transform concept
    on "type"       <concept>:      ConceptType (<formal_name>, <type_spec>, <where>),
    on "value"      <concept>:      ConceptValue (<formal_name>, <formal_arguments>,
                                                <where>, <formal_arguments>, <where>)
;
transform formal_parameters
    on "empty"      <formal_parameters>:  FormalParmEmpty(),
    on "fieldlist"  <formal_parameters>:  FormalParmList (<field_list>, <where>)
;
transform formal_arguments
    on "empty"      <formal_arguments>:  FormalArgEmpty(),
    on "fieldlist"  <formal_arguments>:  FormalArgList (<field_list>)
;
transform field_list
    on "single"     <field_list>:      SingleField (<field>),
    on "multiple"   <field_list>:      MultiField (<field_list>, <field>)
;
transform field
    on "namelist"   <field>:          FieldNameList (<name_list>, <type_spec>),
    on "collection" <field>:          FieldCollection (<identifier>, <type_spec>)
;
transform type_spec
    on "named"      <type_spec>:      NamedType (<actual_name>)
;
transform name_list
    on "single"     <name_list>:      NameIdent (<identifier>),
    on "multiple"   <name_list>:      NamePair (<name_list>, <identifier>)
;

```

```

transform actual_parameters
  on "empty"      <actual_parameters>:  ActParmEmpty(),
  on "arglist"    <actual_parameters>:  ListOfArgs(<arg_list>)
;
transform actual_arguments
  on "empty"      <actual_arguments>:    ActualArgNil(),
  on "arglist"    <actual_arguments>:    ActualArgList(<arg_list>)
;
transform arg_list
  on "single"     <arg_list>:            SingleArg(<arg>),
  on "multiple"   <arg_list>:            MultiArg(<arg_list>,<arg>)
;
transform arg
  on "expression" <arg>:                ArgExp(<expression>),
  on "bindpair"  <arg>:                ArgPair(<pair>)
;
transform expression_list
  on "single"     <expression_list>:    SingleExp(<expression>),
  on "multiple"   <expression_list>:    MultiExp(<expression_list>,<expression>)
;
transform expression
  on "quantify"   <expression>:    QuantifierExp(<quantifier>,<field_list>,
                                                <restriction>,<expression>),
  on "actname"    <expression>:    IdentExp(<actual_name>,<actual_arguments>),
  on "@"          <expression>:    AtExp(<actual_name>,<actual_name>,
                                         <actual_arguments>),
  on "~"          <expression>:    NotExp(<expression>),
  on "&"          <expression>:    AndExp(<expression>,<expression>),
  on "|"          <expression>:    OrExp(<expression>,<expression>),
  on "implies"    <expression>:    ImpliesExp(<expression>,<expression>),
  on "iff"        <expression>:    IffExp(<expression>,<expression>),
  on "<"          <expression>:    LessThan(<expression>,<expression>),
  on ">"          <expression>:    GreaterThan(<expression>,<expression>),
  on "="          <expression>:    Equal(<expression>,<expression>),
  on "<="         <expression>:    LessEqual(<expression>,<expression>),
  on ">="         <expression>:    GreaterEqual(<expression>,<expression>),
  on "~="         <expression>:    NotEqual(<expression>,<expression>),
  on "~<"         <expression>:    NotLessThan(<expression>,<expression>),
  on "~>"         <expression>:    NotGreaterThan(<expression>,<expression>),
  on "~<="        <expression>:    NotLessEqual(<expression>,<expression>),
  on "~>="        <expression>:    NotGreaterEqual(<expression>,<expression>),
  on "=="         <expression>:    Equivalent(<expression>,<expression>),
  on "~=="        <expression>:    NotEquivalent(<expression>,<expression>),
  on "uminus"     <expression>:    UnaryMinus(<expression>),
  on "+"          <expression>:    Add(<expression>,<expression>),
  on "-"          <expression>:    Subtract(<expression>,<expression>),
  on "*"          <expression>:    Multiply(<expression>,<expression>),
  on "/"          <expression>:    Divide(<expression>,<expression>),
  on "mod"        <expression>:    ModExp(<expression>,<expression>),
  on "***"        <expression>:    Exponent(<expression>,<expression>),
  on "U"          <expression>:    UExp(<expression>,<expression>),
  on "append"     <expression>:    AppendExp(<expression>,<expression>),
  on "in_exp"     <expression>:    InExp(<expression>,<expression>),
  on "star"       <expression>:    Star(<expression>),
  on "collect"    <expression>:    CollectExp(<expression>),
  on "range"      <expression>:    RangeExp(<expression>,<expression>),
  on "dot"        <expression>:    DotExp(<expression>,<identifier>),
  on "[]"         <expression>:    SqBracketExp(<expression>,<expression>),
  on "parens"     <expression>:    ParenExp(<expression>),
  on "measure"    <expression>:    MeasureExp(<expression>,<identifier>),
  on "time"       <expression>:    TimeExp(),

```

```

    on "delay"      <expression>: DelayExp(),
    on "period"     <expression>: PeriodExp(),
    on "constant"   <expression>: Constant(<literal>),
    on "lit_type"   <expression>: LiteralType(<literal>,<actual_name>),
    on "illegal"    <expression>: IllegalExp(),
    on "if_then"    <expression>: IfThen(<expression>,<expression>,<middle_cases>,<expression>)
;
transform middle_cases
    on "empty"      <middle_cases>: MidCaseNil(),
    on "middle"     <middle_cases>: MidCase(<middle_cases>,<expression>,<expression>)
;
transform quantifier
    on "all"        <quantifier>: All(),
    on "some"       <quantifier>: Some(),
    on "number"     <quantifier>: Number(),
    on "sum"        <quantifier>: Sum(),
    on "product"    <quantifier>: Product(),
    on "set"        <quantifier>: Set(),
    on "max"        <quantifier>: Maximum(),
    on "min"        <quantifier>: Minimum(),
    on "union"      <quantifier>: Union(),
    on "intersect"  <quantifier>: Intersection()
;
transform restriction
    on "empty"      <restriction>: RestrictNil(),
    on "suchthat"   <restriction>: SuchThat(<expression>)
;
transform literal
    on "int"        <literal>: IntLiteral(<integer_lit>),
    on "real_lit"   <literal>: RealLiteral(<real_lit>),
    on "char_lit"   <literal>: CharLiteral(<char_lit>),
    on "str_lit"    <literal>: StringLiteral(<string_lit>),
    on "enum"       <literal>: Enumeration(<identifier>),
    on "seq"        <literal>: Sequence(<expressions>),
    on "set"        <literal>: SetLiteral(<expressions>),
    on "map"        <literal>: MapLiteral(<expressions>,<expression>),
    on "tuple"      <literal>: TupleLiteral(<pair_list>),
    on "one_of"     <literal>: OneOfLiteral(<pair>)
;
transform expressions
    on "empty"      <expressions>: ExpressionsNil(),
    on "explist"    <expressions>: ExpressionsList(<expression_list>)
;
transform pair_list
    on "single"     <pair_list>: SinglePair(<pair>),
    on "multi"      <pair_list>: MultiPair(<pair_list>,<pair>)
;

```

B. LEXICAL PHYLA DECLARATIONS

The file *lexemes.ssl* contains the declarations of the lexemes for the SPECDEF editor (i.e., the keywords, punctuation marks, and other special characters.

```
/* File: LEXEMES.SSL */
/*
/*      This file contains the lexeme declarations for the SPECDEF editor */

/* lexemes */
WHITESPACE:    Whitespace< [\ \t\n]* >;

AND:           AndLex< "&" >;
OR:            OrLex< "|" >;
NOT:           NotLex< "~" >;
IMPLIES:       ImpliesLex< "=>" >;
IFF:           IffLex< "<=>" >;

GT:            GtLex< ">" >;
LT:            LtLex< "<" >;
LE:            LeLex< "<=" >;
GE:            GeLex< ">=" >;
NE:            NeLex< "~=" >;
NLT:           NltLex< "~<" >;
NGT:           NgtLex< "~>" >;
NLE:           NleLex< "~<=" >;
NGE:           NgeLex< "~>=" >;
EQV:           EqvLex< "==" >;
NEQV:          NeqvLex< "~==" >;

RANGE:         RangeLex< ".." >;
APPEND:        AppendLex< "!" >;
MOD:           ModLex< "\\" >;
EXP:           ExpLex< "*" >;
BIND:          BindLex< "::" >;

IF:            IfLex< "IF" >;
THEN:          ThenLex< "THEN" >;
ELSE:          ElseLex< "ELSE" >;
IN:            InLex< "IN" >;
U:            ULex< "U" >;

ALL:           AllLex< "ALL" >;
SOME:          SomeLex< "SOME" >;
NUMBER:        NumberLex< "NUMBER" >;
SUM:           SumLex< "SUM" >;
PRODUCT:       ProductLex< "PRODUCT" >;
SET:           SetLex< "SET" >;
MAXIMUM:       MaximumLex< "MAXIMUM" >;
MINIMUM:       MinimumLex< "MINIMUM" >;
UNION:         UnionLex< "UNION" >;
INTERSECTION:  IntersectLex< "INTERSECTION" >;
SUCH:          SuchLex< "SUCH"[\ \t\n]+"THAT" >;
ELSE_IF:       ElseIfLex< "ELSE"[\ \t\n]+"IF" >;
```

```

AS:           AsLex< "AS" >;
CONCEPT:    ConceptLex< "CONCEPT" >;
DEFINITION:   DefnLex< "DEFINITION" >;
DELAY:        DelayLex< "DELAY" >;
END:          EndLex< "END" >;
EXPORT:       ExportLex< "EXPORT" >;
FI:           FileLex< "FI" >;
FROM:         FromLex< "FROM" >;
HIDE:         HideLex< "HIDE" >;
IMPORT:       ImportLex< "IMPORT" >;
INHERIT:      InheritLex< "INHERIT" >;
PERIOD:       PeriodLex< "PERIOD" >;
RENAME:       RenameLex< "RENAME" >;
TIME:         TimeLex< "TIME" >;
VALUE:        ValueLex< "VALUE" >;
WHERE:        WhereLex< "WHERE" >;

SEMI:         SemiLex< ";" >;
COMMA:        CommaLex< "," >;
PLUS:         PlusLex< "+" >;
MINUS:        MinusLex< "-" >;
MUL:          MulLex< "*" >;
DIV:          DivLex< "/" >;
UMINUS:       UminusLex< "-" >;
DOT:          DotLex< "." >;
STAR:         StarLex< "*" >;

INTEGER_LIT:  IntegerLex< [0-9]+ >;
REAL_LIT:     RealLex< [0-9]+ "." [0-9]+ >;
CHAR_LIT:     CharLex< "'" [^\n]"' >;
STRING_LIT:   StringLex< "[" [^\n\]"*[" >;
IDENTIFIER:   IdentLex< [a-zA-Z]{a-zA-Z_0-9}* >;

```

C. ATTRIBUTE DECLARATIONS

The file *attribs.ssl* contains the attribute declarations and replacement rules for the concrete input syntax for textual input.

```

/* File: ATTRIBS.SSL                                     */
/*                                                         */
/*      This file contains the attribute declarations for the concrete */
/*      input syntax for the SPECDEF editor.                  */
/*                                                         */

/* association between abstract syntax and concrete input syntax */
Module      { syn module      t; };
Interface   { syn interface   t; };
Inherits    { syn inherits    t; };
Hide        { syn hide        t; };
Renames     { syn renames     t; };
Imports     { syn imports     t; };
Export      { syn export      t; };
Where       { syn where       t; };
Concepts    { syn concepts    t; };
Concept     { syn concept     t; };
FormalName  { syn formal_name t; };

```

```

FormalParm      ( syn formal_parameters t; );
FormalArg       ( syn formal_arguments  t; );
FieldList       ( syn field_list         t; );
Field           ( syn field               t; );
TypeSpec        ( syn type_spec          t; );
NameList        ( syn name_list          t; );
ActualName      ( syn actual_name        t; );
ActualParm      ( syn actual_parameters  t; );
ActualArgs      ( syn actual_arguments   t; );
ArgList         ( syn arg_list           t; );
Arg             ( syn arg                t; );
ExpList         ( syn expression_list    t; );
Exp             ( syn expression         t; );
MiddleCases     ( syn middle_cases       t; );
Quantifier      ( syn quantifier         t; );
Restriction     ( syn restriction         t; );
Literal         ( syn literal            t; );
Expressions     ( syn expressions        t; );
PairList        ( syn pair_list          t; );
Pair            ( syn pair               t; );
IntegerLit      ( syn integer_lit        t; );
RealLit         ( syn real_lit           t; );
CharLit         ( syn char_lit           t; );
StringLit       ( syn string_lit         t; );
Ident           ( syn identifier         t; );

```

/* replacement rules */

```

module          ~      Module.t;
interface       ~      Interface.t;
inherits        ~      Inherits.t;
hide            ~      Hide.t;
renames         ~      Renames.t;
imports         ~      Imports.t;
export          ~      Export.t;
where           ~      Where.t;
concepts        ~      Concepts.t;
concept         ~      Concept.t;
formal_name     ~      FormalName.t;
formal_parameters ~ FormalParm.t;
formal_arguments ~ FormalArg.t;
field_list      ~      FieldList.t;
field           ~      Field.t;
type_spec       ~      TypeSpec.t;
name_list       ~      NameList.t;
actual_name     ~      ActualName.t;
actual_parameters ~ ActualParm.t;
actual_arguments ~ ActualArgs.t;
arg_list        ~      ArgList.t;
arg             ~      Arg.t;
expression_list ~      ExpList.t;
expression      ~      Exp.t;
middle_cases    ~      MiddleCases.t;
quantifier      ~      Quantifier.t;
restriction     ~      Restriction.t;
literal         ~      Literal.t;
expressions     ~      Expressions.t;
pair_list       ~      PairList.t;
pair            ~      Pair.t;

```

```

integer_lit ~ IntegerLit.t;
real_lit   ~ RealLit.t;
char_lit   ~ CharLit.t;
string_lit ~ StringLit.t;
identifier ~ Ident.t;

```

D. PRECEDENCE RULES AND CONCRETE INPUT SYNTAX

The file *concrete.ssl* contains the precedence and associativity rules for the concrete input syntax, plus all the concrete parsing declarations.

```

/* File: CONCRETE.SSL */
/*
/*      This file contains the precedence and associativity declarations
/*      and the concrete input syntax parsing rules for the SPECDEF editor. */

/* precedence and associativity declarations */
left  ';' IF IDENTIFIER SEMI;
left  ',' COMMA;
left  SUCH;
left  IFF;
left  IMPLIES;
left  '||' OR;
left  '&' AND;
left  '~' NOT;
left  '<' '>' '=' LE GE NE NLT NGT NLE NGE EQV NEQV;
nonassoc IN RANGE;
left  'U' APPEND;
left  '+' '-' PLUS MINUS;
left  '*' '/' MUL DIV MOD;
left  UMINUS;
left  EXP;
left  '$' '[' '{' '.' DOI WHERE;
left  STAR;

/* concrete input syntax */
Module      ::= (DEFINITION Interface Concepts END)
              {Module.t = DefModuleDecl(Interface.t, Concepts.t);}

Interface   ::= (FormalName Inherits Imports Export)
              {Interface.t = InterfaceDecl(FormalName.t, Inherits.t,
              Imports.t, Export.t);}

Inherits    ::= () { $$ .t = InheritNil(); }
              | (INHERIT Inherits ActualName Hide Renames)
              { Inherits$1.t = InheritStmt(Inherits$2.t, ActualName.t,
              Hide.t, Renames.t); }

Hide        ::= () { $$ .t = HideNil(); }
              | (HIDE NameList) { Hide.t = HideList(NameList.t); }

Renames     ::= () { $$ .t = RenameNil(); }
              | (Renames RENAME Ident AS Ident)
              { Renames$1.t = RenameStmt(Renames$2.t, Ident$1.t, Ident$2.t); }

```

```

Imports      ::= ()    { $$ . t = ImportNil(); }
              | (Imports IMPORT NameList FROM ActualName)
                { Imports$1.t = ImportStmt(Imports$2.t, NameList.t, ActualName.t); }
              ;

Export       ::= ()    { $$ . t = ExportNil(); }
              | (EXPORT NameList) { Export.t = ExportList(NameList.t); }
              ;

Where        ::= (prec SEMI) { $$ . t = WhereEmpty(); }
              | (WHERE ExpList) { Where.t = WhereExp(ExpList.t); }
              ;

Concepts     ::= (Concept) { Concepts.t = (Concept.t :: ConceptNil); }
              | (Concept Concepts)
                { Concepts$1.t = (Concept.t :: Concepts$2.t); }
              ;

Concept      ::= (CONCEPT FormalName ':' TypeSpec Where)
                { Concept.t = ConceptType(FormalName.t, TypeSpec.t, Where.t); }
              | (CONCEPT FormalName FormalArg Where VALUE FormalArg Where)
                { Concept.t = ConceptValue(FormalName.t, FormalArg$1.t, Where$1.t,
                                           FormalArg$2.t, Where$2.t); }
              ;

FormalName   ::= (Ident FormalParm)
                { FormalName.t = FormName(Ident.t, FormalParm.t); }
              ;

FormalParm   ::= ()    { $$ . t = FormalParmEmpty(); }
              | ('{' FieldList '}' Where)
                { FormalParm.t = FormalParmList(FieldList.t, Where.t); }
              ;

FormalArg    ::= ()    { $$ . t = FormalArgEmpty(); }
              | ('(' FieldList ')')
                { FormalArg.t = FormalArgList(FieldList.t); }
              ;

FieldList    ::= (Field) { FieldList.t = SingleField(Field.t); }
              | (FieldList ',' Field)
                { FieldList$1.t = MultiField(FieldList$2.t, Field.t); }
              ;

Field        ::= ('?') { Field.t =.UndefField(); }
              | (NameList ':' TypeSpec)
                { Field.t = FieldNameList(NameList.t, TypeSpec.t); }
              | ('$' Ident ':' TypeSpec)
                { Field.t = FieldCollection(Ident.t, TypeSpec.t); }
              ;

TypeSpec     ::= ('?') { TypeSpec.t =.UndefType(); }
              | (ActualName) { TypeSpec.t = NamedType(ActualName.t); }
              ;

NameList     ::= (Ident) { NameList.t = NameIdent(Ident.t); }
              | (NameList ',' Ident)
                { NameList$1.t = NamePair(NameList$2.t, Ident.t); }
              ;

ActualName   ::= (Ident ActualParm)
                { ActualName.t = ActualParmList(Ident.t, ActualParm.t); }
              ;

ActualParm   ::= (prec SEMI) { $$ . t = ActParmEmpty(); }
              | ('{' ArgList '}' ) { ActualParm.t = ListOfArgs(ArgList.t); }
              ;

ActualArgs   ::= (prec SEMI) { $$ . t = ActualArgNil(); }
              | ('(' ArgList ')') { ActualArgs.t = ActualArgList(ArgList.t); }
              ;

ArgList      ::= (Arg) { ArgList.t = SingleArg(Arg.t); }
              | (ArgList ',' Arg prec COMMA)
                { ArgList$1.t = MultiArg(ArgList$2.t, Arg.t); }
              ;

```

```

Arg      ::= (Exp)      {Arg.t = ArgExp(Exp.t);}
          | (Pair)      {Arg.t = ArgPair(Pair.t);}
          ;

ExpList  ::= (Exp prec COMMA) (ExpList.t = SingleExp(Exp.t));
          | (ExpList ',' Exp prec COMMA)
            (ExpList$1.t = MultiExp(ExpList$2.t, Exp.t));
          ;

Exp      ::= ('?')      {Exp.t = UndefExp();}
          | (Quantifier '(' FieldList Restriction BIND Exp ')')
            (Exp$1.t = QuantifierExp(Quantifier.t, FieldList.t,
                                     Restriction.t, Exp$2.t));
          | (ActualName ActualArgs) {Exp.t = IdentExp(ActualName.t, ActualArgs.t);}
          | (ActualName '@' ActualName ActualArgs)
            {Exp.t = AtExp(ActualName$1.t, ActualName$2.t, ActualArgs.t);}
          | ('~' Exp prec NOT)      {Exp$1.t = NotExp(Exp$2.t);}
          | (Exp 's' Exp prec AND)   {Exp$1.t = AndExp(Exp$2.t, Exp$3.t);}
          | (Exp '|' Exp prec OR)    {Exp$1.t = OrExp(Exp$2.t, Exp$3.t);}
          | (Exp IMPLIES Exp prec IMPLIES)
            {Exp$1.t = ImpliesExp(Exp$2.t, Exp$3.t);}
          | (Exp IFF Exp prec IFF)   {Exp$1.t = IffExp(Exp$2.t, Exp$3.t);}
          | (Exp '<' Exp prec LE)     {Exp$1.t = LessThan(Exp$2.t, Exp$3.t);}
          | (Exp '>' Exp prec LE)     {Exp$1.t = GreaterThan(Exp$2.t, Exp$3.t);}
          | (Exp '=' Exp prec '=')   {Exp$1.t = Equal(Exp$2.t, Exp$3.t);}
          | (Exp LE Exp prec LE)     {Exp$1.t = LessEqual(Exp$2.t, Exp$3.t);}
          | (Exp GE Exp prec LE)     {Exp$1.t = GreaterEqual(Exp$2.t, Exp$3.t);}
          | (Exp NE Exp prec LE)     {Exp$1.t = NotEqual(Exp$2.t, Exp$3.t);}
          | (Exp NLT Exp prec LE)    {Exp$1.t = NotLessThan(Exp$2.t, Exp$3.t);}
          | (Exp NGT Exp prec LE)
            {Exp$1.t = NotGreaterThan(Exp$2.t, Exp$3.t);}
          | (Exp NLE Exp prec LE)    {Exp$1.t = NotLessEqual(Exp$2.t, Exp$3.t);}
          | (Exp NGE Exp prec LE)
            {Exp$1.t = NotGreaterEqual(Exp$2.t, Exp$3.t);}
          | (Exp EQV Exp prec LE)    {Exp$1.t = Equivalent(Exp$2.t, Exp$3.t);}
          | (Exp NEQV Exp prec LE)   {Exp$1.t = NotEquivalent(Exp$2.t, Exp$3.t);}
          | (UMINUS Exp prec UMINUS) {Exp$1.t = UnaryMinus(Exp$2.t);}
          | (Exp '-' Exp prec PLUS)  {Exp$1.t = Add(Exp$2.t, Exp$3.t);}
          | (Exp '-' Exp prec MINUS) {Exp$1.t = Subtract(Exp$2.t, Exp$3.t);}
          | (Exp '*' Exp prec MUL)   {Exp$1.t = Multiply(Exp$2.t, Exp$3.t);}
          | (Exp '/' Exp prec DIV)   {Exp$1.t = Divide(Exp$2.t, Exp$3.t);}
          | (Exp MOD Exp prec MOD)   {Exp$1.t = ModExp(Exp$2.t, Exp$3.t);}
          | (Exp EXP Exp prec EXP)   {Exp$1.t = Exponent(Exp$2.t, Exp$3.t);}
          | (Exp U Exp prec U)       {Exp$1.t = UExp(Exp$2.t, Exp$3.t);}
          | (Exp APPEND Exp prec APPEND) {Exp$1.t = AppendExp(Exp$2.t, Exp$3.t);}
          | (Exp IN Exp prec IN)     {Exp$1.t = InExp(Exp$2.t, Exp$3.t);}
          | ('*' Exp prec STAR)      {Exp$1.t = Exp$2.t;}
          | ('$' Exp prec DOT)       {Exp$1.t = CollectExp(Exp$2.t);}
          | (Exp RANGE Exp prec RANGE) {Exp$1.t = RangeExp(Exp$2.t, Exp$3.t);}
          | (Exp '.' Ident prec DOT) {Exp$1.t = DotExp(Exp$2.t, Ident.t);}
          | (Exp '[' Exp ']' prec DOT) {Exp$1.t = SqBracketExp(Exp$2.t, Exp$3.t);}
          | ('(' Exp ')')            {Exp$1.t = ParenExp(Exp$2.t);}
          | ('(' Exp Ident ')')      {Exp$1.t = MeasureExp(Exp$2.t, Ident.t);}
          | (TIME)                   {Exp.t = TimeExp();}
          | (DELAY)                   {Exp.t = DelayExp();}
          | (PERIOD)                   {Exp.t = PeriodExp();}
          | (Literal)                 {Exp.t = Constant(Literal.t);}
          | (Literal '@' ActualName)
            {Exp.t = LiteralType(Literal.t, ActualName.t);}
          | ('!')                     {Exp.t = IllegalExp();}
          | (IF Exp THEN Exp MiddleCases ELSE Exp FI)
            {Exp$1.t = IfThen(Exp$2.t, Exp$3.t, MiddleCases.t, Exp$4.t);}
          ;

```

```

MiddleCases ::= () { $$t = MidCaseNil(); }
| (MiddleCases ELSE_IF Exp THEN Exp)
| (MiddleCases$1.t = MidCase(MiddleCases$2.t, Exp$1.t, Exp$2.t));
;

Quantifier ::= (ALL) { $$t = All(); }
| (SOME) { $$t = Some(); }
| (NUMBER) { $$t = Number(); }
| (SUM) { $$t = Sum(); }
| (PRODUCT) { $$t = Product(); }
| (SET) { $$t = Set(); }
| (MAXIMUM) { $$t = Maximum(); }
| (MINIMUM) { $$t = Minimum(); }
| (UNION) { $$t = Union(); }
| (INTERSECTION) { $$t = Intersection(); }
;

Restriction ::= () { $$t = RestrictNil(); }
| (SUCH Exp) { Restriction.t = SuchThat(Exp.t); }
;

Literal ::= (IntegerLit) { Literal.t = IntLiteral(IntegerLit.t); }
| (RealLit) { Literal.t = RealLiteral(RealLit.t); }
| (CharLit) { Literal.t = CharLiteral(CharLit.t); }
| (StringLit) { Literal.t = StringLiteral(StringLit.t); }
| ('#' Ident) { Literal.t = Enumeration(Ident.t); }
| ('[' Expressions ')') { Literal.t = Sequence(Expressions.t); }
| ('{' Expressions '}') { Literal.t = SetLiteral(Expressions.t); }
| ('{' Expressions ';' Exp '}') { Literal.t = MapLiteral(Expressions.t, Exp.t); }
| ('[' PairList ']') { Literal.t = TupleLiteral(PairList.t); }
| ('{' Pair '}') { Literal.t = OneOfLiteral(Pair.t); }
;

Expressions ::= () { $$t = ExpressionsNil(); }
| (ExpList) { Expressions.t = ExpressionsList(ExpList.t); }
;

PairList ::= (Pair) { PairList.t = SinglePair(Pair.t); }
| (PairList ',' Pair) { PairList$1.t = MultiPair(PairList$2.t, Pair.t); }
;

Pair ::= (NameList BIND Exp) { Pair.t = PairBind(NameList.t, Exp.t); }
;

IntegerLit ::= (INTEGER_LIT) { IntegerLit.t = IntConstant(INTEGER_LIT); }
;

RealLit ::= (REAL_LIT) { RealLit.t = RealConstant(REAL_LIT); }
;

CharLit ::= (CHAR_LIT) { CharLit.t = CharConstant(CHAR_LIT); }
;

StringLit ::= (STRING_LIT) { StringLit.t = StringConstant(STRING_LIT); }
;

Ident ::= (IDENTIFIER) { Ident.t = Identifier(IDENTIFIER); }
;

```

E. UNPARSING SCHEMES

The file *unparse.ssl* contains the unparsing declarations for the display representation of the constructs for the SPECDEF editor.

```

/* File: UNPARSE.SSL                                     */
/*                                                       */
/*      This file contains the unparsing rules for the SPECDEF editor.      */
/*                                                       */

/* Unparsing Scheme */
start      :      Spec          [ @ : @ ]
;
spec       :      ModuleNil     [ @ : ]
;
           :      ModulePair   [ @ ::= ^ ["%n%n"] @ ]
;
module     :      DefModuleDecl [ @ ::= "DEFINITION " @ @ "%n"
                                "END" ]
;
interface  :      InterfaceDecl [ @ ::= @ "%t" @ @ @ "%b%n" ]
;
innerits   :      InheritNil    [ ^ : ]
           :      InheritPrompt [ ^ ::= "<inherit>" ]
           :      InheritStmnt [ ^ ::= @ "%n" "INHERIT " @ "%t" @ @ "%b" ]
;
hide       :      HideNil       [ ^ : ]
           :      HidePrompt    [ ^ ::= "<hide>" ]
           :      HideList      [ ^ ::= "%n" "HIDE " @ ]
;
renames    :      RenameNil     [ ^ : ]
           :      RenamePrompt  [ ^ ::= "<rename>" ]
           :      RenameStmnt   [ ^ ::= @ "%n" "RENAME " @ " AS " @ ]
;
imports    :      ImportNil     [ ^ : ]
           :      ImportPrompt  [ ^ ::= "<import>" ]
           :      ImportStmnt   [ ^ ::= @ "%n" "IMPORT " @ " FROM " @ ]
;
export     :      ExportNil     [ ^ : ]
           :      ExportPrompt  [ ^ ::= "<export>" ]
           :      ExportList    [ ^ ::= "%n" "EXPORT " @ ]
;
where      :      WhereEmpty    [ ^ : ]
           :      WherePrompt   [ ^ ::= "<where>" ]
           :      WhereExp      [ ^ ::= "WHERE " @ ]
;
concepts   :      ConceptNil    [ @ ::= ]
           :      ConceptPair   [ @ ::= ^ @ ]
;
concept    :      ConceptComp   [ @ ::= " <concept>" ]
           :      ConceptType   [ ^ ::= "%t%n" "CONCEPT " @ " : " @ "%t%n"
                                @ "%b%n" ]
           :      ConceptValue  [ ^ ::= "%t%n" "CONCEPT " @ " " @ " " @ "%t%n"
                                "VALUE " @ "%n" @ "%b%n" ]
;
formal_name :      FormName     [ ^ ::= @ " " @ ]
;

```



```

NotLessEqual    [ ^ ::= "%{" @ "%{to ~<= " @ "%}" ]
NotGreaterEqual [ ^ ::= "%{" @ "%{to ~>= " @ "%}" ]
Equivalent      [ ^ ::= "%{" @ "%{to == " @ "%}" ]
NotEquivalent    [ ^ ::= "%{" @ "%{to ~= " @ "%}" ]
Unaryminus      [ ^ ::= "-" @ ]
Add             [ ^ ::= @ " + " @ ]
Subtract        [ ^ ::= @ " - " @ ]
Multiply         [ ^ ::= @ " * " @ ]
Divide          [ ^ ::= @ " / " @ ]
ModExp          [ ^ ::= @ " \ " @ ]
Exponent        [ ^ ::= @ " ^ " @ ]
UExp            [ ^ ::= "%{" @ "%{to U " @ "%}" ]
AppendExp       [ ^ ::= "%{" @ "%{to || " @ "%}" ]
InExp           [ ^ ::= "%{" @ "%{to IN " @ "%}" ]
Star            [ ^ ::= "*" @ ]
CollectExp      [ ^ ::= "$" @ ]
RangeExp        [ ^ ::= @ ".." @ ]
DotExp          [ ^ ::= @ "." @ ]
SqBracketExp    [ ^ ::= @ "[" @ "]" ]
ParenExp        [ ^ ::= "(" @ ")" ]
MeasureExp      [ ^ ::= "(" @ @ ")" ]
TimeExp         [ ^ : "TIME" ]
DelayExp        [ ^ : "DELAY" ]
PeriodExp       [ ^ : "PERIOD" ]
Constant        [ ^ ::= @ ]
LiteralType     [ ^ ::= @ "@" @ ]
IllegalExp      [ ^ : "!" ]
IfThen          [ ^ ::= "%{" IF " @ "%to%b THEN " @ "%to%b"
                  @ "%to%b ELSE " @ " FI" } ]

/
middle_cases : MidCaseNil    [ ^ : ]
              MiddlePrompt [ ^ ::= "<middle_cases>" ]
              MidCase      [ ^ ::= "%{" @ "%to ELSE IF " @ "%to THEN " @ "%!" ]

/
quantifier : QuantifierComp [ ^ ::= "<quantifier>" ]
            All             [ ^ : "ALL" ]
            Some            [ ^ : "SOME" ]
            Number          [ ^ : "NUMBER" ]
            Sum             [ ^ : "SUM" ]
            Product         [ ^ : "PRODUCT" ]
            Set             [ ^ : "SET" ]
            Maximum         [ ^ : "MAXIMUM" ]
            Minimum         [ ^ : "MINIMUM" ]
            Union           [ ^ : "UNION" ]
            Intersection     [ ^ : "INTERSECTION" ]

/
restriction : RestrictComp [ ^ ::= "<restriction>" ]
            RestrictNil    [ ^ ::= ]
            SuchThat       [ ^ ::= "SUCH THAT " @ ]

/
literal : EmptyLiteral [ ^ ::= "<literal>" ]
        IntLiteral    [ ^ ::= @ ]
        RealLiteral   [ ^ ::= @ ]
        CharLiteral   [ ^ ::= @ ]
        StringLiteral [ ^ ::= @ ]
        Enumeration    [ ^ ::= "@" @ ]
        Sequence       [ ^ ::= "{" @ "}" ]
        SetLiteral     [ ^ ::= "{" @ "}" ]

```

```

      MapLiteral      [ ^ ::= "{" @ " ; " @ " }" ]
      TupleLiteral    [ ^ ::= "{" @ " }" ]
      OneOfLiteral     [ ^ ::= "{" @ " }" ]
;
expressions          : ExpressionsComp [ ^ ::= "<expressions>" ]
                      | ExpressionsNil [ ^ ::= ]
                      | ExpressionsList [ ^ ::= @ ]
;
pair_list             : PairListComp   [ ^ ::= "<pair_list>" ]
                      | SinglePair     [ ^ ::= @ ]
                      | MultiPair       [ ^ ::= @ " , " @ ]
;
pair                  : PairBind       [ ^ ::= @ " :: " @ ]
;
integer_lit           : IntConstant    [ ^ ::= ^ ]
;
real_lit              : RealConstant    [ ^ ::= ^ ]
;
char_lit              : CharConstant    [ ^ ::= ^ ]
;
string_lit            : StringConstant  [ ^ ::= ^ ]
;
identifier            : Identifier      [ ^ ::= ^ ]
;

```

APPENDIX C

I. USER'S MANUAL FOR THE SPECDEF EDITOR

The SPECDEF editor is a *syntax directed editor* for writing Spec specification language definition modules. A syntax directed editor, also known as a *language-based editor*, is one in which the programs and systems are created and modified according to the *syntactic structure* of the language. The programs are represented internally as *abstract syntax trees*, or *parse trees*, that are built by the editor through constructive commands entered by the user.

The "programs" that are manipulated in the SPECDEF editor are actually specifications, not compilable programs. However, the Spec language has a very well-defined syntactic structure, just like most programming languages, so a syntax directed editor works just as well for Spec as it does for any structured programming language.

The SPECDEF editor uses the principle of *immediate computation* to analyze the syntactic correctness of the abstract syntax tree as it is being constructed. This means that as the user enters each additional component to the parse tree, or modifies an existing component, the entire tree is re-analyzed to see if the addition or modification maintains a correct syntactic structure. If it does not, the user is immediately signalled that a syntax error has occurred, which must be corrected before the editor will allow the user to add any more components. In this way, the editor ensures that only syntactically correct (i.e.,

error-free) specifications are written. This is the key feature of any syntax directed editor.

This editor was created using the Synthesizer Generator, a CASE tool for generating syntax directed editors from an editor specification file containing the grammar rules (the abstract syntax), the display representation (known as the unparsing scheme), and the concrete input syntax rules of the language for which the editor is designed, in this case Spec.

A. STANDARD FEATURES

Each editor specification results in an editor with distinct characteristics that are language-dependent. However, there is a generic user interface common to all editors created by the Synthesizer Generator. A few of the interface features have minor differences dependent on the windowing system for which the editor is targeted. These differences are indicated where applicable.

Throughout this user's manual, editor system commands are written in **boldface**. Many of these commands have one or more *key-bindings*, which are indicated by the text between angle brackets (e.g., < ^I or ESC-x >). These are keys on the keyboard that will invoke the commands. The symbol ^ refers to the CTRL key, which must be held down while depressing the key indicated immediately following the ^ symbol. For example, ^C means "hold down the CTRL key and depress the C key".

The display screen of the editor will normally have some portion of the displayed text highlighted in reverse-video. In the diagrams that follow, characters that would appear in reverse-video on the actual display are written in **boldface**.

1. Invoking and Exiting the Editor

The editor is invoked by typing the name SPECDEF (or SPECDEF_SUN for the Sun workstation version of the editor) at the operating system prompt with an optional list of arguments consisting of one or more filenames of files to be loaded into the editing buffers. Note that the editor name must be typed in all capital letters. Invoking the editor with no arguments causes editing to begin in the default buffer main with no associated file. [Ref. 11: p. 90]

To terminate an editing session, leave the editor and return to the operating system, the exit command < ^C, ESC-^C, or ^X^C > is executed. [Ref. 11: p. 91]

2. Display Screen

The editor's display screen is divided into four horizontal stripes, or regions, labelled from top to bottom, the *title bar*, the *command line*, the *object pane*, and the *help pane*, as shown in Figure 22. The title bar is always highlighted and contains the name of the current buffer displayed in the object pane. The command line echos commands and error messages. The object pane displays all or a fragment of the current buffer's contents. The help pane, which takes up the last few lines of the display window, lists the currently selected constituent, or node in the parse tree, plus the currently enabled template transformations, if any. [Ref. 10: p. 21; 11: p. 91]

It should be noted that the horizontal lines separating the panes in all the diagrams of this user's manual are for explanation purposes only. They do not appear on the actual video display terminal. The entire title bar, however, is always highlighted.

Title Bar
Command Line
Object Pane
Help Pane

Figure 22
Display screen regions

On the Sun workstation version of the editor, the window will also have two scroll bars for use with the mouse: one to the right of the object pane and one below the object pane. These scroll bars control which portion of the buffer is visible in the object pane. [Ref. 11: p. 91]

3. Current Selection vs. Locator

SPECDEF is a screen-oriented hybrid structure editor, since it permits editing the structure of the displayed object through template insertions as well as allowing character- and line-oriented operations on the text. A *template* of a component consists of a pattern of keywords and *placeholders* where additional components can be inserted. A Spec specification is created top down by invoking *template transformations* that insert new components within the framework of previously entered templates. These template transformations are the constructive commands that actually build the abstract syntax tree. [Ref. 10: pp. 21,26]

The current *selection* indicates the individual component of the specification at which the editor is positioned. This is indicated on the screen display by highlighting all components contained within the template of the current selection. The help pane also tells the user which component is currently selected by displaying the message "Positioned at *<component>*", where *<component>* is the phylum name of the current selection. (A *phylum* can be thought of as a node in the abstract syntax tree.) [Ref. 10: p. 21]

A *character selection* is displayed during text editing, indicating the specific character position within the highlighted selection where the next character can be entered. On standard video display terminals, this position is denoted by an unhighlighted character within the highlighted selection; on Sun workstations, the character selection is indicated by an I-beam symbol within the highlighted selection. [Ref. 10: pp. 22,26]

The *locator* can be used to change the current selection and the character selection. On standard video display terminals, the locator is the terminal's cursor, generally denoted by a blinking underbar or a blinking solid box. The locator can be moved with the commands **pointer-up** *< ESC-p >*, **pointer-down** *< ESC-d >*, **pointer-left** *< ESC-b >*, and **pointer-right** *< ESC-f >*. Even after the locator is moved to a new component, the new selection is not made until the **select** command *< ESC-@ >* is executed.

- **CAUTION:** The locator does *not* automatically move to the new selection if the user changes selection by executing the **forward-with-optionals** command **< ^M or RETURN >**. This can be very confusing to the user, since the cursor position will not correspond to the actual position in the parse tree structure of the object being edited.

On Sun workstations, the locator is denoted by an arrow. The locator's position is changed by moving the mouse, and the **select** command is executed by clicking the mouse's selection button (the left button). [Ref. 10: pp. 22,300]

B. TEXT ENTRY

Selections are displayed according to the unparsing schemes, or display templates, declared in the editor specification. Each production in the grammar has its own template. The template for each selection is either editable as text or it is immutable. Note that a selection declared as immutable may have elements within its template that are not immutable. The selection must be moved to the editable part before text entry is permitted. [Ref. 10: pp. 22-23]

If the current selection is not editable as text, any attempt to type a character is rejected by the editor, sounds a warning signal, and displays the message "text entry not permitted here" on the command line, as shown in Figure 23. The selection must be changed to an editable component by either repositioning the locator and executing the **select** command **< ESC-@ >**, or by executing one of the other structural selection by traversal commands, such as **forward-preorder < ^N >**, **backward-preorder < ^P >**, **forward-with-optionals < ^M or RETURN >**, **forward-sibling < ESC-^N >**, **forward-sibling-with-optionals < ESC-^M >**, etc. (See Chapter 3 of [Ref. 11] for other structural selection by traversal commands.) [Ref. 10: pp. 22-24; 11: pp. 101-102]

Current buffer : main
text entry not permitted here
DEFINITION
END
Positioned at spec

Figure 23
Example of illegal attempt to enter text

When the current selection is editable, the entered text is captured into the *text buffer*, which is displayed in its proper position in the object pane. During editing, the selection does not exist as structure, but instead exists as text. Operations within the selection are defined on individual characters rather than on the structure. The text entry is terminated by executing **forward-with-optionals** < ^M or RETURN >, at which time syntactic correctness is checked. [Ref. 10: p. 26]

If a syntax error is detected, a warning signal sounds, the message "syntax error" appears on the command line, and the character selection is positioned at the last character of the leftmost error that was detected. For example, in Figure 24, an attempt was made to enter (p : passenger) as a formal argument to concept waiting, but the left and right parentheses were not entered, resulting in a syntax error. [Ref. 10: p. 27]

Current buffer : main		
syntax error		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting p : passenger END		
Positioned at formal_arguments	empty	fieldlist

Figure 24
Example of a syntax error

To correct a syntax error, the user can either delete the entire text buffer by executing the command **delete-selection** < ^K > and then entering the correct text, or the text buffer itself can be edited with any of a variety of commands described below. There will be one character highlighted within the text buffer. This character selection can be moved with the commands **right** < ^F >, **left** < ^B >, **beginning-of-line** < ^A >, or **end-of-line** < ^E >. Once the character selection is at the desired position, the user can either type additional characters to be inserted, or delete unwanted characters. Deletion commands are **delete-next-character** < ^D > (which actually deletes the currently highlighted character), **delete-previous-character** < DEL > (which deletes the character to the left of the cursor), **erase-to-end-of-line** < ESC-d >, or **erase-to-beginning-of-line** < ESC-DEL >. [Ref. 11: pp. 103-104,109]

C. TEMPLATE INSERTION

When a new selection is made, the help pane is updated to indicate which component is selected and any *template transformations* that are enabled. The template names listed are actually commands used to invoke the respective transformations. [Ref. 10: p. 23]

Templates may be selected in three ways, dependent on which type of terminal is being used. On standard video display terminals, the user must escape to the command line by invoking the **execute-command** command `< ^I, TAB, or ESC-x >`, and then typing a sufficient number of characters of the desired template name to make the choice unambiguous. If the prefix is ambiguous because an insufficient number of characters are typed, the message "`<prefix> is ambiguous`" appears on the command line, where `<prefix>` is the string that the user entered, as shown in Figures 25 and 26. On Sun workstations or other mouse-equipped workstations, transformations can be invoked by clicking on the *transformation-name* in the help pane, or by choosing from a pop-up menu of choices with the mouse's structure-menu button (the right button). [Ref. 10: p. 23; 11: p. 147]

In the SPECDEF editor, it is recommended to expand the size of the help pane by two lines to be able to view all the possible transformations available for the phylum expression. This is accomplished by executing the **enlarge-help** `< ESC-^Xz >` command twice (once for each line of expansion). The default size of the help pane is four lines, but phylum expression requires six lines, as shown in Figures 25 and 26. [Ref. 11: p. 98]

Current buffer : main							
COMMAND: if							
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(p : passenger , f : floor ?) <concept> END							
Positioned at expression		quantify	actname	@	~	&	
	implies	iff	<	>	=	<=	>=
~=	~<	~>	~<=	~>=	==	~==	uminus
+	-	*	/	mod	**	U	append
in_exp	star	collect	range	dot	[]	parens	measure
time	delay	period	constant	lit_type	illegal	if_then	

Figure 25
Ambiguous template transformation selection

Current buffer : main							
if is ambiguous							
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(p : passenger , f : floor ?) <concept> END							
Positioned at expression		quantify	actname	@	~	&	
	implies	iff	<	>	=	<=	>=
~=	~<	~>	~<=	~>=	==	~==	uminus
+	-	*	/	mod	**	U	append
in_exp	star	collect	range	dot	[]	parens	measure
time	delay	period	constant	lit_type	illegal	if_then	

Figure 26
Result of ambiguous transformation selection

Some selections do not have any associated template transformations. This is reflected in the help pane by listing only the current constituent. If an attempt is made to initiate a template insertion when no transformations are enabled, or an attempt is made to select a transformation other than those enabled, as in Figure 27, the editor will sound a warning signal, display the message "<template-name> is unknown command", where <template-name> is the character string typed by the user, and a help buffer will be displayed listing all the system commands and their associated key-bindings available at that location, as in Figure 28. The help buffer can be removed from the screen with the delete-other-window command < ^X1 >. [Ref. 10: p. 25]

Current buffer : main						
COMMAND: forall						
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE <quantifier>(<field><restriction> :: ?) <concept> END						
Positioned at	quantifier		all	some	number	sum
set	max	min	union	intersect		product

Figure 27
Attempt to initiate non-available transformation

The **forward-with-optionals** command < ^M or RETURN > advances the selection forward to the next component in preorder when executed alone, including stopping at any optional placeholder inserted by the editor, such as in a list. However, when entering

text or invoking a template transformation, the **forward-with-optionals** command behaves differently. When a transformation is invoked, the selection moves to the first placeholder (excluding optionals) in the transformed component. If no such placeholder exists, then the selection remains at the same position where the transformation was invoked. In such instances, it appears to the user that the **forward-with-optionals** command must be executed twice to advance to the next selection. [Ref. 10: pp. 37-38]

Current buffer : main						
forall is unknown command						
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger						
Positioned at quantifier set	max	min	all union	some intersect	number	sum product
Current buffer : Help						
advance-after-transform				(none)		
advance-after-parse				(none)		
apropos				<ESC ?>		
ascend-to-parent				<ESC \>		
backward-preorder				<^P>		
backward-sibling				<ESC ^P>		
backward-sibling-with-optionals				<ESC ^B>		
backward-with-optionals				<^H>		
beginning-of-file				<ESC <>		
beginning-of-line				<^A>		

Figure 28
Unknown Command message and Help buffer

D. SAVING AN EDITING SESSION

Edited objects are contained in a named *buffer*, such as the main buffer. To save the object being edited, the buffer contents must be written to a file. The command used to write such a file depends on whether the object was created directly during the current editing session, or if it was a previously existing file loaded into the buffer as an argument file when the editor was invoked, or was loaded with the command **read-file** *file-name* < ^X^F or ^X^R >. [Ref. 11: 95-97]

There are two file *formats* in which an object may be saved: *text* or *structure*. A text file contains the display representation of the object as it would be viewed in the object pane. A structure file contains the internal representation of the edited object. [Ref. 11: p. 95]

An object that was saved as a structure file can be read back into the editor for re-editing. The unparsing schemes that were in effect at the time the file was written are restored, since they are saved as part of the file in these formats. A text file, however, can only be read into the editor if it can be reparsed, because the unparsing schemes in effect at the time the file was written are not saved as part of the file. If the text file contains syntactic errors, the file is read into the text buffer and the character selection is positioned near the error. This would occur in the case where the user saved a file immediately after attempting a syntactically incorrect text entry but before correcting the error. [Ref. 11: p. 96]

When a previously existing file is read into a buffer, the format of the file becomes associated with the buffer for subsequent **write-current-file** commands < ^Xs >. This

coinmand writes the current modified contents of the buffer back to the same file using the same format as when the file was read into the buffer. [Ref. 11: p. 97]

If the edited object was created during the current editing session, or if the user desires to save a previously existing file in a different format, execute the command **write-named-file** *file-name format* < ^X^W >. The default format is structure. A pop-up window appears at the bottom of the display screen. This window contains parameter fields for the user to enter the filename of the file to save the buffer contents into, and the format in which to save it. When the user wants to save his work for later re-editing, the structure format should be selected. If the user wants to print the contents of the buffer, the file should be saved as a text file, which can then be printed as any regular text file. An example of this pop-up window is shown in Figures 29 and 30. Once the parameters are selected, the command **start-command** < ESC-s > must be executed to actually write the contents to the file. When the editor is done writing the file, the message "Wrote <filename>" is displayed on the command line, where <filename> is the name of the file being written, and the parameters window disappears. [Ref. 11: pp. 93,97]

Current buffer : main
<pre> DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(<field><restriction> :: ?) <concept> END </pre>
<pre> ---Current Form : Write File Form----- WRITE FILE: <file-name> FILE FORMAT: structure Positioned at _file_name ----- </pre>

Figure 29
Pop-up write-named-file parameter window

Current buffer : main
<pre> DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(<field><restriction> :: ?) <concept> END </pre>
<pre> ---Current Form : Write File Form----- COMMAND: text WRITE FILE: pass_def.spec FILE FORMAT: structure Positioned at _file_type text structure ----- </pre>

Figure 30
Specifying text format for Write File Form

E. EDITOR SYSTEM COMMANDS

Many of the editor system commands are similar or identical to EMACS text editor command names and standard key-bindings [Ref. 11: p. 90]. The more commonly used commands have been described in the preceeding sections and are summarized below. There are numerous other less commonly used commands. Appendix D contains a complete list of the available editor commands. For a full description of each command, the reader is referred to Chapter 3 of *The Synthesizer Generator Reference Manual* [Ref. 11].

1. Executing Commands and Transformations

Every editor command has a *name* and zero or more *key-bindings*. There are four ways to invoke a command. First, the keystrokes bound to the command may be typed. Second, the command may be selected from a menu. Third, the command name may be typed on the command line of a window. And lastly, on mouse-equipped systems only, a mouse click may select an actuator bound to the command. All keystrokes other than the command key-bindings or escaped text entered on the command line are interpreted as textual insertions into the object being edited. [Ref. 11: p. 91]

The following commands are used to execute transformations or to get out of the editor:

- **exit** < ^C, ESC-^C, ^X^C >: Leave the editor and return to the shell. If any buffers have been modified since their associated files were last written out, a warning is issued in a pop-up window. To abort the **exit**, execute as **cancel-command** < ESC-c >. To continue with the **exit** command, execute a **start-command** < ESC-s >.

- **execute-command** *name* < ^I, ESC-x >: Initiates the command mode and redirects subsequent characters to the command line following the prompt **COMMAND:**. Entry on the command line is terminated by the first blank, carriage return, or other command key-binding. The command name need not be typed in its entirety; any prefix of a command that uniquely identifies the command is sufficient. Transformations take priority over built-in commands. If the command has no parameters, it is executed immediately. Otherwise, a pop-up window for parameters appears; after the parameters have been provided, the command should be initiated by executing **start-command**. If a command parameter is either invalid or ambiguous, an error message is issued. Note that the control character ^I is TAB.
- **start-command** < ESC-s >: Initiate execution of a command with the parameters contained in the current pop-up parameter form window. If not currently editing a parameter form, **start-command** does nothing.

2. Buffers, Selections, and Files

Objects that are being manipulated in the editor are contained in a collection of named *buffers*. Each file being edited is typically read into a distinct buffer. The buffer is then associated with the given file until either a different file is read into that buffer, or the given buffer is written out to a different file. [Ref. 11: p. 95]

Two file *formats* are supported: *text* and *structure*. Text files contain the display representation of a term. Structure files contain an internal representation of a term. A *term* is a derivation tree of the object being manipulated. [Ref. 11: p. 95]

Terms written as structure files can be read back into the editor with the identical structure as when the file was saved. Text files, however, can only be read back into the editor if the text can be reparsed because the unparsing schemes in effect at the time the file was written are not saved within the file. [Ref. 11: p. 96]

The following commands are used to read or write files:

- **read-file** *file-name* < ^X^F, ^X^R >: Reads the named file into the current buffer, deleting the previous contents of that buffer. The buffer becomes associated with that file for subsequent **write-current-file** commands. If the current buffer is already associated with a file that has not been written since the buffer was last updated, you must answer **yes** to the question **Overwrite buffer?** before the read will be executed. If the given file is text, it must be syntactically correct with respect to the input syntax of the given editor. If syntactically incorrect, the file is read into the text buffer with the cursor positioned near the error. If the given file is a structured file, the term contained in the file and the term currently contained in the buffer must be in the same phylum.
- **write-current-file** < ^Xs >: Write the value of the buffer displayed in the current window to its associated file in the current format associated with the buffer.
- **write-named-file** *file-name format* < ^X^W >: Write the value of the buffer displayed in the current window to the given file in the given format. The default format--structure--can be changed by selecting the *format* field of the pop-up parameter window and invoking the text transformation.

3. Creating, Deleting, and Resizing Windows and Panes

In editors generated for standard video display terminals, windows are non-overlapping, are arranged in horizontal stripes across the screen, and are cyclicly ordered from top to bottom for the purpose of the **next-window** and **previous-window** commands. [Ref. 11: p. 97]

In editors generated for workstations with high-resolution, bitmapped displays (e.g., Sun workstations), resizable and overlapping windows with scroll bars are supported. [Ref. 11: p. 98]

The following commands manipulate the windows and display:

- **delete-other-windows** < ^X1 >: Delete all windows other than the current one.
- **next-window** < ^Xn >: Switch to the next window on the screen. (Video display terminals only.)

- **previous-window** < ^Xp >: Switch to the previous window on the screen. (Video display terminals only.)
- **enlarge-help** < ESC-^Xz >: Increase the size of the help pane of the current window by one line.
- **redraw-display** < ^L >: Refresh the screen image to remove any spurious characters.

4. Changing the Structural Selection by Traversal of the Edited Term

The current selection can be changed by moving through the structure of the abstract-syntax tree (e.g., in preorder, reverse preorder, by moving to the parent, etc.). If such a motion would cause the selection to leave the currently displayed text in the window, the object is scrolled automatically to keep the new selection within the window. [Ref. 11: p. 101]

Some commands cause the editor to automatically insert instances of optional components into the abstract syntax tree (e.g., **forward-with-optionals** < ^M or RETURN >). The placeholder for an optional element becomes the current selection. If the selection is advanced beyond an inserted optional element, and that optional element is still only a placeholder, that placeholder is removed from the abstract syntax tree (and consequently from the display). Other commands that change the selection ignore any optional components and therefore do not insert an instance of such elements in the abstract syntax tree. [Ref. 11: p. 101]

The following commands are used to move the selection:

- **forward-preorder** < ^N >: Change the selection to the next resting place in a forward preorder traversal of the abstract-syntax tree. Do not stop at placeholders for optional constituents.

- **backward-preorder** < ^P >: Change the selection to the previous resting place in a forward preorder traversal of the abstract-syntax tree. Do not stop at placeholders for optional constituents.
- **right** < ^F >: If there is no text buffer, **right** is the same as **forward-preorder**. See Section I.E.5 for the meaning of **right** when there is a text buffer.
- **left** < ^B >: If there is no text buffer, **left** is the same as **backward-preorder**. See Section I.E.5 for the meaning of **left** when there is a text buffer.
- **forward-with-optionals** < ^M >: Change the selection to the next resting place in a forward preorder traversal of the abstract-syntax tree. Stop at placeholders for optional constituents. Note the ^M is RETURN.
- **end-of-file** < ESC-> >: Change the selection to the rightmost resting place in the abstract syntax tree.

5. Changing the Character Selection by Traversal of the Text Buffer

The text buffer contains text being entered or re-edited. This buffer is displayed in its proper position within its enclosing structural context. One character within the text buffer is selected, shown by a highlight. This character selection can be moved within the text buffer by normal horizontal and vertical "cursor-motion" commands. If the character selection is moved beyond the boundaries of the text buffer, the contents of the text buffer are submitted for syntactic analysis and translation. Within the text buffer, the structural-motion commands **forward-preorder**, **backward-preorder**, **right**, and **left**, which were previously defined for moving the component selection, have been overloaded and are also used to move the character selection down, up, right, and left, respectively. [Ref. 11: p. 103]

The following commands are used to change the character selection within the text buffer:

- **forward-preorder** < ^N >: Move the character selection one position down. If already at the last line of the text buffer, this command is interpreted as **forward-after-parse**.
- **backward-preorder** < ^P >: Move the character selection one position up. If already at the first line of the text buffer, this command is interpreted as **backward-preorder**, as described in Section I.E.4, provided the text is syntactically correct.
- **right** < ^F >: Move the character selection one position to the right. If already at the rightmost character of a line, the character selection advances to the first character of the next line of the text buffer. If already at the rightmost character of the last line of the text buffer, the command is interpreted as **forward-after-parse**.
- **left** < ^B >: Move the character selection one position to the left. If already at the leftmost character of a line, the character selection advances to the last character of the previous line of the text buffer. If already at the leftmost character of the first line of the text buffer, the command is interpreted as **backward-preorder**.
- **beginning-of-line** < ^A >: Move the character selection to the beginning of the line.
- **end-of-line** < ^E >: Move the character selection to the end of the line.
- **forward-after-parse** (no key-binding): If textual entry is terminated by **forward-preorder**, then upon successful analysis the **forward-preorder** command is replaced by **forward-after-parse**. Let t be the subterm or sublist that has replaced the selection as a result of textual input. If no existing placeholder occurs within t , then **forward-after-parse** stops at the first resting place beyond t . **Forward-after-parse** never inserts optional placeholders either in t or beyond t .

6. Moving the Locator on the Screen

The commands described in this section apply only to editors generated for standard video display terminals (for editors generated for workstations equipped with a mouse, the mouse can be used to point anywhere in the object pane, and clicking the

mouse will change the current selection). It is important to note that the locator is distinct from the selection. The locator identifies a point on the screen and not a point in the buffer (i.e., the locator does not necessarily agree with the selection in the parse tree).

[Ref. 11: p. 105]

The following commands move the locator (i.e., the cursor):

- **pointer-left < ESC-b >:** Move the locator one character to the left. If already in column one of the object pane and not already at the leftmost scroll position, scroll the window.
- **pointer-right < ESC-f >:** Move the locator one character to the right. If already at the right border of the object pane, scroll the window.
- **pointer-up < ESC-p >:** Move the locator one character up. If already at the top of the object pane and not already at the uppermost scroll position, scroll the window.
- **pointer-down < ESC-d >:** Move the locator one character down. If already at the bottom of the object pane, scroll the window.

7. Structural Editing

Structural modifications follow a cut-and-paste paradigm, similar to block-edit functions found in many text editors. Only whole, well-formed substructures can be removed and inserted. [Ref. 11: p. 108]

The following commands are used to move, copy, or delete entire subtrees of the overall abstract syntax tree:

- **cut-to-clipped < ^W >:** Move the selection of the current buffer to the distinguished buffer CLIPPED. The removed selection is replaced by a placeholder, which becomes the new selection. The previous contents of CLIPPED are lost.
- **copy-to-clipped < ESC-^W >:** Copy the selection of the current buffer to buffer CLIPPED. The previous contents of CLIPPED are lost.

- **paste-from-clipped < ^Y >**: Move the contents of buffer CLIPPED into the buffer at the current selection, which must be a placeholder. In CLIPPED, a placeholder term replaces the previous contents.
- **copy-from-clipped < ESC-^Y >**: Copy the contents of buffer CLIPPED into the buffer at the current selection, which necessarily must be a placeholder. The contents of CLIPPED are left unchanged.
- **copy-text-from-clipped < ESC-^T >**: Copy the contents of buffer CLIPPED, as text, into a text buffer at the current selection immediately preceding the character selection. The contents of CLIPPED are left unchanged.
- **delete-selection < ^K >**: Move the selection of the current buffer to the distinguished buffer DELETED. The selection becomes a placeholder. The previous contents of DELETED are lost.

8. Textual Editing

Textual insertion and textual re-editing are permitted for some components of the language, as discussed in Section I.E.5 above.

If a textual insertion is permitted at a placeholder, you merely begin to type; this causes the text of the placeholder to disappear and the keystrokes are echoed in the text buffer, which is displayed in place on the screen.

If textual re-editing of an existing structure is desired and is permitted, the character selection is positioned at the desired place in the text buffer whereupon the user can begin to either type or erase characters. If the current selection was established by tree traversal, as described in Section I.E.4 above, then the textual-insertion point is in front of the character at which the locator pointed when the **select-start** was executed.

[Ref. 11: p. 109]

The following commands are used to edit text within the text buffer:

- **delete-next-character** < ^D >: Delete the current character selection. If the character selection is at the end of a line in the text buffer (other than the last line), then the current line and the next line are joined into one line.
- **delete-previous-character** < DEL >: Delete the character to the left of the character selection. If the character selection is at the beginning of a line in the text buffer (other than the first line), then the current line and the previous line are joined into one line.
- **erase-to-end-of-line** < ESC-d >: Erase from the character selection to the end of the line, including the character selection.
- **erase-to-beginning-of-line** < ESC-DEL >: Erase from the beginning of the line to the character selection, not including the character selection.
- **delete-selection** < ^K >: Delete the entire line.

F. SAMPLE EDITING SESSION

We present a sample editing session using the standard video display terminal version of SPECDEF. We will create a small specification file, save it for later re-editing, recall the file into the editor, modify it, and save it for printout.

Some terminology that is used throughout the remainder of this section needs to be clarified:

- Recall that a *phylum* can be thought of as a node in the abstract syntax tree.
- A *completing term* is the default representation of a phylum. Completing terms can be replaced through template transformations, if any are enabled. The completing term for some phyla are user prompts, consisting of the phylum name enclosed in angle brackets, e.g., <formal_parm>. Other phyla have template patterns with keywords for the respective phylum.
- A *placeholder term* is also a default representation of a phylum. For non-optional phyla, the placeholder term is identical to the completing term. But for phyla that are declared as optional, the placeholder is usually a user prompt, as described above, indicating where a transformation can be inserted if desired.

In the diagrams that follow, recall that text that would appear as highlighted text on an actual video display is printed in **boldface**.

We invoke the editor by typing SPECDEF at the Unix prompt with no arguments, as we will be creating the specification in this editing session. The initial screen presented to the user has a blank object pane, as shown in Figure 31. Note that the help pane indicates the selection is positioned at phylum start.

Current buffer : main
Positioned at start

Figure 31
Initial Display Screen

There are no template transformations enabled for phylum start, so we simply advance the selection by executing **forward-with-optionals** < ^M or RETURN >. After depressing the RETURN key, we are presented with the display as shown in Figure 32. Phylum spec is declared as an optional list phylum, since we may have zero or more occurrences of this construct according to the Spec grammar. Therefore, what we see in the display is the placeholder term for phylum spec, which happens to be the completing term for phylum module, the first argument for phylum spec.

Current buffer : main
DEFINITION
END
Positioned at spec

Figure 32
Advancing the selection to phylum spec

As we advance the selection by executing **forward-with-optionals** < ^M or **RETURN** > repeatedly to cycle through phyla module, interface, and formal_name, the only changes that occur to the display screen are the name of the current selection listed in the help pane, and the disappearance of the highlighting of the keywords **DEFINITION** and **END** as the selection reaches **interface**. The next selection is **identifier**, at which point we decide to enter the name of our module, **waiting_passenger**, as shown in Figure 33.

Current buffer : main
DEFINITION waiting_passenger
END
Positioned at identifier

Figure 33
Entering text at node identifier

The text entry is terminated by entering a carriage return. This causes the entered text to be checked for syntactic correctness; it is accepted and inserted in the abstract-syntax tree at the node for phylum identifier. The selection is advanced to `formal_parameters`, as shown in Figure 34.

Here we have the first occurrence of a completing term/placeholder term/user prompt, the string `<formal_parm>`, to prompt the user that some acceptable form of formal parameters should be entered here. We also have two template transformations enabled named `empty` and `fieldlist`. In this case we do not want any formal parameters, so we invoke the transformation `empty` by first executing `execute-command < ^I or TAB >`; when we depress the TAB key, the prompt `COMMAND:` appears on the command line. We only need to enter enough letters of the transformation name to make our choice unambiguous; in this case, simply typing `e` is sufficient to differentiate between `empty` and `fieldlist`. After typing the `e`, the screen appears as in Figure 35.

Current buffer : main		
DEFINITION waiting_passenger<formal_parm>		
END		
Positioned at formal_parameters	empty	fieldlist

Figure 34
Display after parsing the module name

Current buffer : main		
COMMAND: e		
DEFINITION waiting_passenger<formal_parm>		
END		
Positioned at formal_parameters	empty	fieldlist

Figure 35
Invoking template transformation **empty**

Any of the several commands that move the selection will terminate the template command. We use **forward-with-optionals** < ^M or RETURN >. The template for an empty formal_parameter list is executed, which replaces the placeholder term <formal_parm> with a null.

The selection is advanced, and the optional phylum **inherits** is selected. Since this phylum is optional, the completing term (a null statement) is replaced on the display with the placeholder term, the string <inherit>. There are two **INHERIT** clauses we wish to add, so we invoke **execute-command** twice, each time typing a for **addinherit** to display the templates for two **INHERIT** clauses, as shown in Figure 36. Note that the selection is still positioned at the <inherit> prompt on the first line, since the abstract-syntax rule for **inherits** inserts an instance of **inherits** at the beginning of each clause until a null clause is selected. To get rid of this prompt and advance to the first **INHERIT** clause, we must invoke the transformation **empty** on the command line.

Current buffer : main		
DEFINITION waiting_passenger<Inherit> INHERIT <hide><rename> INHERIT <hide><rename> END		
Positioned at inherits	empty	addinherit

Figure 36
After invoking transformation addinherit twice

After invoking transformation empty, the selection is advanced to **actual_name**. We type in the name **elevator**, the name of the module we wish to inherit. Even though we did not include any actual parameters after the actual name **elevator**, it is still accepted, since actual parameters can be empty. Note that we could have advanced the selection one level lower, to phylum identifier, before entering the name **elevator**, but then we would also have had to invoke the transformation for an empty actual parameter list and an empty actual argument list. By entering the text at phylum **actual_name**, we have saved some time and effort by not having to bother with the empty parameter and argument constructs. Our display now appears as in Figure 37.

We do not wish to hide or rename any concepts from the inherited module **elevator**, so for each of those selections in turn we invoke the **empty** template transformation the same way we did before.

Current buffer : main		
DEFINITION waiting_passenger INHERIT elevator <hide><rename> INHERIT <hide><rename> END		
Positioned at hide	empty	addhide

Figure 37
After entering inherited module name elevator

Note that after invoking **execute-command** and entering an **e** followed by a carriage return, the help pane still indicates the selection is positioned at **hide** (also the same for **rename**). This is because the **hide** (**rename**) that was replaced by a null statement was the optional selection that was inserted by the editor when we executed **forward-with-optionals**. When the null statement fills that node, the selection moves back up the abstract-syntax tree to the **hide** (**rename**) within the template for **inherits**. By simply executing **forward-with-optionals** < ^M or RETURN >, the placeholder <hide> (<rename>) is replaced by the completing term (~ null statement) and the selection is advanced.

The same process is performed for the second **INHERIT** clause, with the name **passenger** for the inherited module. Again, we do not wish to hide or rename any concepts from the inherited module **passenger**, so we invoke the **empty transformation** for each of these constructs.

The selection is now positioned at **imports**, another optional phylum. The placeholder term replaces the completing term on the display, and our screen looks like that shown in Figure 38.

Current buffer : main		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger <import> END		
Positioned at imports	empty	addimport

Figure 38
Selection positioned at phylum imports

We do not have any concepts we wish to import, so we invoke the **empty** transformation. As we saw with **phyla hide** and **rename**, we must execute **forward-with-optionals** after completing the transformation to advance the selection to **export**. We will not export any concepts, so the identical process is performed for **export** as we did for **imports**.

The selection is now positioned at **concepts**, which is declared as an optional list phylum. The completing term for the first argument of **concepts** is displayed, which is the prompt **<concept>** from phylum **concept**, as shown in Figure 39. Note that the help pane indicates we are positioned at **concepts**, but we still have two template transformations enabled, **type** and **value**, which are associated with phylum **concept**. This

feature minimizes the distinction between a singleton sublist and the list item itself, and is true for any list phylum. [Ref. 10: p. 67]

Current buffer : main		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger <concept> END		
Positioned at concepts	type	value

Figure 39
Prompt for entering a concept

We wish to add a concept that returns a value, so we invoke the **execute-command** command < ^I or TAB > and enter v and a carriage return to invoke the value transformation. We are presented with the template for a **concept** that returns a value, as shown in Figure 40.

The selection is positioned at **formal_name**, the first argument for phylum concept. After executing **forward-with-optionals**, the selection is positioned at **identifier**. We enter the name of our concept, **waiting**, and are presented with the display as shown in Figure 41.

We do not have any formal parameters, so we invoke the transformation **empty**. However, we do have a single argument to concept **waiting**. With the selection at **formal_arguments**, we invoke the transformation **fieldlist**. This leaves the selection positioned at **field_list** with transformation choices **single** and **multiple**. We invoke transformation **single** and are presented with the display as shown in Figure 42.

Current buffer : main
<pre> DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT <formal_args> <where> VALUE <formal_args> <where> END </pre>
Positioned at formal_name

Figure 40
Template for a concept returning a value

Current buffer : main
<pre> DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting<formal_parm><formal_args> <where> VALUE <formal_args> <where> END </pre>
Positioned at formal_parameters empty fieldlist

Figure 41
Display after entering the concept name waiting

Current buffer : main		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting(?) <where> VALUE <formal_args> <where> END		
Positioned at field	namelist	collection

Figure 42
After invoking a single field_list

As we did with actual_name, we will enter the complete text required to make up a field, which consists of a name_list and a type_spec. We type **p : passenger** (including the space either side of the colon) and enter a carriage return. Note that we could have invoked the transformation namelist, which would have necessitated our entering the name_list and type_spec separately. As before, we elected to enter the complete construct at the field node to save time and keystrokes.

The selection is now positioned at the first <where> prompt, with two transformations enabled, empty and addwhere. The display appears as in Figure 43.

Current buffer : main		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting(p : passenger) <where> VALUE <formal_args> <where> END		
Positioned at where	empty	addwhere

Figure 43
After entering the formal argument (p : passenger)

There is no WHERE clause that applies to the formal arguments, so we invoke **execute-command**, type **e** and a carriage return to invoke the **empty** transformation. We must also execute **forward-with-optionals** to remove the <where> prompt and advance to the next selection.

The selection is now positioned at the second occurrence of formal_arguments, the prompt <formal_args> after the keyword VALUE. Two transformations are enabled, empty and fieldlist. We invoke the fieldlist transformation, which replaces the string <formal_args> with (<field>). The help pane indicates the current selection is field_list with two transformations enabled, single and multiple. We invoke the single transformation, and are presented with a screen as shown in Figure 44. Note that the completing term/placeholder term for a field (i.e., ?) has replaced the <field> prompt.

Current buffer : main		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting(p : passenger) VALUE (?) <where> END		
Positioned at field	namelist	collection

Figure 44
After invoking the single transformation for field_list

Just as before, we can enter the entire field construct of a `name_list` and a `type_spec` separated by a free-standing colon. We enter **b : boolean** and a carriage return to fill the `field` node of the parse tree.

At this point, we are called away by our boss to a meeting and we wish to save our work to resume later. We execute the command **write-named-file < ^X^W >**; a pop-up window overlays the bottom of the display screen for us to enter the *filename* and *format* parameters to the **write-named-file** command, as shown in Figure 45.

The selection is positioned at `_file_name` in the Write File Form window. We type in the desired name (including any extension); for this example we choose the name **waitpass.spec**. Just as in the object pane, we terminate text entry by entering a carriage return. The selection advances to `_file_type` with two transformations enabled,

text and structure. The default file format, structure, is already highlighted in the **FILE FORMAT:** field. If you plan on doing more editing on a file, the recommended format in which to save the file is structure. Text format is used to save the display as a printable file. Since the default format is the one we want, no further entry is required. To complete the execution of the **write-named-file** command with our selected parameters, we must execute **start-command < ESC-s >**. This causes the command line to echo "Wrote waitpass.spec", and the actual file to be written to the file system.

Current buffer : main
<pre> DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting(p : passenger) VALUE (b : boolean) <where> END </pre>
<pre> ---Current Form : Write File Form----- WRITE FILE: <file-name> FILE FORMAT: structure Positioned at _file_name </pre>

Figure 45
Pop-up Write File Form window

Note that we are still in the editor. To **exit** the editor, we simply type **^C**, which deactivates the editor and returns us to the Unix shell.

To recall our saved file back into the editor for further editing, we have two choices: we can include the filename **waitpass.spec** as an argument to **SPECDEF** when starting

up the editor, or we can start SPECDEF with no arguments and then execute the **read-file** command `< ^X^R or ^X^F >`, which will cause a pop-up window to appear, similar to the Write File Form window when we saved the file, in which we enter the filename. In this example, we choose the former method.

The initial display this time includes the filename **waitpass.spec** in the title bar, the message "Read waitpass.spec" on the command line, as much of the structure that will fit in the object pane (in this case the entire file), and the indication in the help pane that we are positioned at start. Figure 46 shows this display.

Current buffer : waitpass.spec
Read waitpass.spec
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting(p : passenger) VALUE (b : boolean) <where> END
Positioned at start

Figure 46
Reading an existing file into the editor

There are, again, two choices on how to proceed: we can either advance the selection through the entire parse tree using **forward-preorder** `< ^N >` or **forward-with-optionals** `< ^M or RETURN >`, or we can use **end-of-file** `< ESC-> >` to position the selection at the rightmost leaf in the parse tree. We choose the latter option to save time,

which now positions the selection where we left off during the previous editing session, at the second `<where>` prompt within the construct for `concept`.

We invoke the transformation `addwhere`, which replaces the placeholder `<where>` with the template for a `WHERE` clause. The selection is advanced to `expression_list` with two transformations enabled, `single` and `multiple`. We invoke the `single` transformation, as shown in Figure 47.

Current buffer : waitpass.spec		
COMMAND: s		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting(p : passenger) VALUE (b : boolean) WHERE <expression_list> END		
Positioned at expression_list	single	multiple

Figure 47
Invoking the `single` transformation for `expression_list`

The selection is now positioned at `expression`. In the `SPECDEF` editor, `phylum` `expression` has more template transformations available than can be listed in the default size of the help pane (4 lines). To be able to see all possible transformation names, the help pane must be enlarged by two lines. This is done by executing the `enlarge-help` command `< ESC-^Xz >` twice, once on each line of enlargement. When this is done, the screen will appear as in Figure 48. Notice that many of the transformation names are

symbolic rather than alpha-numeric. Note also the completing/placeholder term for expression, the symbol ?, which stands for an undefined expression.

Current buffer : waitpass.spec							
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ? END							
Positioned at expression			quantify	actname	@	~	&
	implies	iff	<	>	=	<=	>=
~=	~<	~>	~<=	~>=	==	~==	uminus
+	-	*	/	mod	**	U	append
in_exp	star	collect	range	dot	[]	parens	measure
time	delay	period	constant	lit_type	illegal	if_then	

Figure 48
Expanded Help Pane for expression's transformations

The expression we wish to enter is a quantified logic expression, so we begin by invoking the quantify transformation. The placeholder ? is replaced by the template for a quantified expression, as shown in Figure 49.

We invoke the all transformation, which replaces the placeholder <quantifier> with the lexeme ALL and advances the selection to **field_list**. In this example, we want a multiple **field_list**, so we invoke the multiple transformation. The resulting display appears as in Figure 50.

Current buffer : waitpass.spec						
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE <quantifier>(<field><restriction> :: ?) END						
Positioned at quantifier	all	some	number	sum	product	
set	max	min	union	intersect		

Figure 49
After invoking the quantify transformation for expression

Current buffer : waitpass.spec		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(<field>,?<restriction> :: ?) END		
Positioned at field_list	single	multiple

Figure 50
After selecting multiple transformation for field_list

We now address each element of the field_list separately. For the first element, which is itself a field_list, we now invoke the single transformation, which replaces the placeholder <field> with the symbol ? and positions the selection at field.

As we have done previously, we will enter the entire construct here to save time and keystrokes; we enter **p : passenger** followed by a carriage return. This enters the text and advances the selection to the second placeholder for phylum field, the second ?. Here we enter **f : floor** and a carriage return. The selection is now advanced to restriction, as shown in Figure 51.

Current buffer : waitpass.spec		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(p : passenger , f : floor<restriction> :: ?) END		
Positioned at restriction	empty	suchthat

Figure 51
After entering a multiple field_list

We do not have any restrictions on these elements, so we invoke the **empty** transformation. This places the selection at **expression** following the bind (::) symbol.

The expression we wish to have apply here is a combination of several "and'd" expressions that imply a predicate expression. We start by invoking the **&** transformation. This replaces the ? with the string ? & ?. We are still positioned at **expression**, so we now invoke the **actname** transformation by executing the **execute-command** command < ^I or TAB > and then typing **ac**, which is sufficient to make our choice unambiguous. The first placeholder for expression is replaced by the placeholder

<actual_arg>, and the selection is advanced to actual_name. We enter the name of the predicate we wish to apply here, **waiting**. The selection is advanced to actual_arguments, with two transformations enabled, empty and arglist. We invoke the arglist transformation and are presented with the display as shown in Figure 52.

Current buffer : waitpass.spec		
DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(p : passenger , f : floor :: waiting(<arg>) & ?) END		
Positioned at arg_list	single	multiple

Figure 52
After invoking arglist transformation for actual_arguments

We enter the single argument p. We then advance the selection to expression using forward-with-optionals < ^M or RETURN >, and invoke the & transformation once more. Proceeding similarly to the sequence described above, we enter the various text elements until we have built the expression at(p , f).

Advancing the selection again to expression, we now invoke the implies transformation. With the selection positioned at the ? on the left side of the => symbol, we first invoke the ~ transformation (the symbol ~ means "not"). Then we proceed similarly to the sequence stated previously until we have built the expression ~button_lit(f).

Once more advancing the selection, we now complete the expression on the right side of the $\Rightarrow$ symbol until we have built the expression `pushes_button(p , f)`. Our concept `waiting` is now complete, as shown in Figure 53.

Current buffer : waitpass.spec		
<pre> DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(p : passenger , f : floor :: waiting(p) & at(p, f) & ~button_lit(f) => pushes_button(p, f)) <concept> END </pre>		
Positioned at concepts	type	value

Figure 53
The completed concept `waiting`

Following a similar procedure, we add a second concept, `pushes_button`. The complete module `waiting_passenger` now looks like Figure 54.

We are now ready to save our module for printout. We invoke the **write-named-file** command `< ^X^W >` as before. The current filename `waitpass.spec` already fills the `_file_name` field because the filename is associated with the current buffer, so we simply enter a carriage return. This time we want to save the file in text format so we can print it out. We invoke **execute-command** `< ^I or TAB >` and type `t` for the text transformation. Finally, we execute **start-command** `< ESC-s >` to actually write the file.

Current buffer : waitpass.spec		
<pre> DEFINITION waiting_passenger INHERIT elevator INHERIT passenger CONCEPT waiting (p : passenger) VALUE (b : boolean) WHERE ALL(p : passenger , f : floor :: waiting(p) & at(p, f) & ~button_lit(f) => pushes_button(p, f)) CONCEPT pushes_button(p : passenger, f : floor) VALUE (b : boolean) END </pre>		
Positioned at where	empty	addwhere

Figure 54
The complete module waiting_passenger

Since we used the same filename as the previously saved structure formatted file, the contents of that file are overwritten by the text format of the current buffer. Now we can exit the editor by executing the exit command < ^C >.

The file waitpass.spec is now saved as a standard ASCII file that appears the same as the display in the object pane. The file can be printed as you would any text file.

If the reader is still unsure about any of the commands that were used in this sample editing session, review Section I.E on the commonly used commands. For a complete list of all commands available with full descriptions, see Chapter 3 of *The Synthesizer Generator Reference Manual* [Ref. 11: pp. 90-111].

APPENDIX D

LIST OF EDITOR COMMANDS

The following is a complete list of all the available editor commands for any editor created by the Synthesizer Generator. Most of the commands have an associated key-binding. Those that don't have a key-binding are indicated by (none). See Chapter 3 of *The Synthesizer Generator Reference Manual* for a full description of each command.

[Ref. 11]

advance-after-transform	(none)
advance-after-parse	(none)
apropos	< ESC-? >
ascend-to-parent	< ESC-\ >
backward-preorder	< ^P >
backward-sibling	< ESC-^P >
backward-sibling-with-optionals	< ESC-^B >
backward-with-optionals	< ^H >
beginning-of-file	< ESC-< >
beginning-of-line	< ^A >
cancel-command	< ESC-c >
column-left	(none)
column-right	(none)
copy-from-clipped	< ESC-^Y >
copy-text-from-clipped	< ^T >
copy-to-clipped	< ESC-^W >
cut-to-clipped	< ^W >
delete-next-character	< ^D >
delete-other-windows	< ^X1 >
delete-previous-character	< DEL >
delete-selection	< ^K >
delete-window	< ^Xd >
dump-off	(none)
dump-on	(none)

break-to-debugger	(none)
alternate-unparsing-on	(none)
alternate-unparsing-toggle	< ESC-e >
end-of-file	< ESC-> >
end-of-line	< ^E >
enlarge-help	< ESC-^Xz >
enlarge-window	< ^Xz >
erase-to-beginning-of-line	< ESC-DEL >
erase-to-end-of-line	< ESC-d >
execute-command	< ^I >
execute-monitor-command	< ^XI >
exit	< ^C >
extend	< ESC-(>
extend-start	(none)
extend-stop	(none)
extend-transition	< ESC-X >
forward-after-parse	(none)
forward-preorder	< ^N >
forward-sibling	< ESC-^N >
forward-sibling-with-optionals	< ESC-^M >
forward-with-optionals	< ^M >
help-off	(none)
help-on	(none)
illegal-operation	< ^G >
insert-file	< ^X^I >
left	< ^B >
list-buffers	< ^X^B >
new-buffer	(none)
new-line	< ^J >
next-line	< ^Z >
next-page	< ^V >
next-window	< ^Xn >
page-left	< ESC-{ >
page-right	< ESC-} >
paste-from-clipped	< ^Y >
pointer-bottom-of-screen	< ESC-. >
pointer-down	< ESC-n >
pointer-left	< ESC-b >
pointer-long-down	(none)
pointer-long-left	(none)
pointer-long-right	(none)
pointer-long-up	(none)
pointer-right	< ESC-f >

pointer-top-of-screen
 pointer-up
 previous-line
 previous-page
 previous-window
 read-file
 redraw-display
 repeat-command
 return-to-monitor
 right
 scroll-to-bottom
 scroll-to-top
 search-forward
 search-reverse
 select
 select-start
 select-stop
 selection-to-left
 selection-to-top
 select-transition
 set-parameters
 show-attribute
 shrink-help
 shrink-window
 spill
 split-current-window
 start-command
 switch-to-buffer
 text-capture
 undo
 alternate-unparsing-off
 visit-file
 write-attribute
 write-current-file
 write-file-exit
 write-modified-files
 write-named-file
 write-selection-to-file

< ESC-, >
 < ESC-p >
 < ESC-z >
 < ESC-v >
 < ^Xp >
 < ^X^R >
 < ^L >
 < ESC-r >
 < ^_ >
 < ^F >
 (none)
 (none)
 < ESC-^F >
 < ESC-^R >
 < ESC-@ >
 (none)
 (none)
 (none)
 < ESC-! >
 < ESC-t >
 (none)
 (none)
 < ESC-^X^Z >
 < ^X^Z >
 (none)
 < ^X2 >
 < ESC-s >
 < ^Xb >
 (none)
 < ^X^U >
 (none)
 < ^X^V >
 (none)
 (none)
 < ^X^F >
 < ^X^M >
 < ^X^W >
 (none)

REFERENCES

1. Berzins, V., and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1989.
2. Fisher, A., *CASE, Using Software Development Tools*, John Wiley & Sons, Inc., 1988.
3. MacLennan, B., *Principles of Programming Languages: Design, Evaluation, and Implementation*, 2nd edition, Holt, Rinehart & Winston, 1987.
4. Booch, G., *Software Engineering with Ada*, 2nd edition, Benjamin/Cummings, 1987.
5. Allison, L., "Syntax Directed Program Editing", *Software--Practice and Experience*, v.13, pp. 453-465, 1983.
6. Morris, J., and Schwartz, M., "The Design of a Language-Directed Editor for Block-Structured Languages", *SIGPLAN Notices*, v. 16, n. 6, pp. 28-33, June 1981.
7. Notkin, D., "The GANDALF Project", *The Journal of Systems and Software*, v. 5, n. 2, pp. 91-105, May 1985.
8. Ellison, R., and Staudt, B., "The Evolution of the GANDALF System", *The Journal of Systems and Software*, v. 5, n. 2, pp. 107-119, May 1985.
9. Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, v. 24, n. 9, pp. 563-573, September 1981.
10. Reps, T., and Teitelbaum, T., *The Synthesizer Generator. A System for Constructing Language-Based Editors*, Springer-Verlag, 1988.
11. Reps, T., and Teitelbaum, T., *The Synthesizer Generator Reference Manual*, 3rd edition, Springer-Verlag, 1989.
12. Naval Postgraduate School Technical Report NPS52-87-033, *Specifying Large Software Systems in Spec*, by V. Berzins and Luqi, pp. 1-11, July 1987.
13. Naval Postgraduate School Technical Report NPS52-89-029, *A Student's Guide to SPEC*, by V. Berzins and R. Kopas, May 1989.
14. Naval Postgraduate School Technical Report NPS52-87-032, *The Semantics of Inheritance in Spec*, by V. Berzins and Luqi, July 1987.

15. Kopas, R., *The Design and Implementation of a Specification Language Type Checker*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1989.
16. Weigand, J., *Design and Implementation of a Pretty Printer for the Functional Specification Language SPEC*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1988.
17. Lesk, M., and Schmidt, E., *Lex--A Lexical Analyzer Generator*, Computer Science Technical Report 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.
18. Johnson, S., *YACC--Yet Another Compiler Compiler*, Bell Laboratories, Murray Hill, New Jersey, July 1978.

INITIAL DISTRIBUTION LIST

- | | | |
|----|---|---|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22304-6145 | 2 |
| 2. | Dudley Knox Library
Code 0142
Naval Postgraduate School
Monterey, California 93943-5002 | 2 |
| 3. | Ada Joint Program Office
OUSDRE (R&AT)
The Pentagon
Washington, D.C. 20301 | 1 |
| 4. | Commanding Officer
Naval Research Laboratory
Code 5150
Attn. Dr. Elizabeth Wald
Washington, D.C. 20375-5000 | 1 |
| 5. | Defense Advanced Research Projects Agency (DARPA)
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Jacob Schwartz
1400 Wilson Boulevard
Arlington, Virginia 22209-2308 | 1 |
| 6. | Defense Advanced Research Projects Agency (DARPA)
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Squires
1400 Wilson Boulevard
Arlington, Virginia 22209-2308 | 1 |
| 7. | Defense Advanced Research Projects Agency (DARPA)
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308 | 1 |
| 8. | Defense Advanced Research Projects Agency (DARPA)
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308 | 1 |

9. Dr. Amiram Yehudai 1
Tel Aviv University
School of Mathematical Sciences
Department of Computer Science
Tel Aviv, Israel 69978

10. Fleet Combat Direction Systems Support Activity 1
Attn. Mike Reiley, Code 00T
San Diego, California 92147-5081

11. Fleet Combat Direction Systems Support Activity 1
Attn. George Roberson, Code 8D
San Diego, California 92147-5081

12. International Software Systems Inc. 1
12710 Research Boulevard, Suite 301
Attn. Dr. R. T. Yeh
Austin, Texas 78759

13. Kestrel Institute 1
Attn. Dr. C. Green
1801 Page Mill Road
Palo Alto, California 94304

14. LT Robert Kopas, USN 1
Department Head School Class 110
Surface Warfare Officers School Command
Newport, Rhode Island 02841-5012

15. Massachusetts Institute of Technology 1
Department of Electrical Engineering and Computer Science
545 Tech Square
Attn. Dr. B. Liskov
Cambridge, Massachusetts 02139

16. Massachusetts Institute of Technology 1
Department of Electrical Engineering and Computer Science
545 Tech Square
Attn. Dr. J. Guttag
Cambridge, Massachusetts 02139

17. MCC AI Laboratory 1
Attn. Dr. Michael Gray
3500 West Balcones Center Drive
Austin, Texas 78759

18. Office of Naval Research 1
Computer Science Division, Code 1133
Attn. Dr. R. Wachter
800 N. Quincy Street
Arlington, Virginia 22217-5000

- | | | |
|-----|---|---|
| 19. | Office of the Secretary of Defense
R & AT/S & CT, RM 3E114
STARS Program Office
Washington, D.C. 20301 | 1 |
| 20. | Oregon Graduate Center
Portland (Beaverton)
Attn. Dr. R. Kieburtz
Portland, Oregon 97005 | 1 |
| 21. | Software Group, MCC
9430 Research Boulevard
Attn. Dr. L. Belady
Austin, Texas 78759 | 1 |
| 22. | University of Pittsburgh
Department of Computer Science
Attn. Dr. Alfs Berztiss
Pittsburgh, Pennsylvania 15260 | 1 |
| 23. | Prof. V. Berzins
Code 52Bz
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100 | 5 |
| 24. | LCDR David H. Beebe, USN
USS CORAL SEA (CV-43)
FPO New York, New York 09550-2720 | 1 |